CSE 20312
Recursion, Divide and Conquer, merge sort
Recursion
Recursion: Review

Function or algorithm is defined in terms of itself (or similar subtasks)

Components:

1. **Base Case**: Non-recursively defined values that limit or cut off recursion

2. **Recursive Step**: Recursive definition or reapplication of function on new subset
Divide and Conquer
Divide and Conquer: Review

Top-down technique that consists of dividing the problem into smaller subproblems and then combining the partial solutions.

**Divide** problem into smaller subproblems

**Conquer** the subproblems by solving them recursively

**Combine** the partial solutions to get an overall solution
Divide and Conquer: Binary Tree Min

Given a binary tree, determine the minimum value in the tree:
Divide and Conquer: Binary Tree Min

template<typename T>
T minimum_tree(Node<T> *root) {
    T minimum = root->data;

    if (root->left) {
        minimum = min(minimum, minimum_tree(root->left));
    }

    if (root->right) {
        minimum = min(minimum, minimum_tree(root->right));
    }

    return minimum;
}

Recursive algorithms implicitly use the process stack and thus can use $O(n)$ space!
Divide and Conquer: **Binary Tree Height**

Given a **binary tree**, determine the height of the tree:
template <typename T>
size_t height_tree(Node<T> *root) {
    if (root == nullptr) {
        return 0;
    }

    size_t left = height_tree(root->left);
    size_t right = height_tree(root->right);

    return max(left, right) + 1;
}
Merge Sort
Merge Sort: Algorithm

1. **Divide** the elements to be sorted into two groups of equal (or almost equal) size

2. **Sort** each of these smaller groups of elements (recursively)

3. **Merge** the two sorted groups into one large sorted list
Merge Sort: Example
Merge Sort: Pseudo-Code

MergeSort(a, n):
    # Base case
    if n <= 1:
        return

    # Partition array in half
    left = n / 2
    right = n - left

    # Sort each partition
    MergeSort(a, left)
    MergeSort(a + left, right)

    # Merge each partition
    Merge(a, left, right)

Merge(a, left, right):
    # Create temporary array
    merged = new [left + right]

    # Merge both arrays into temporary array
    for l = 0, r = 0, i = 0; i < left + right; i++:
        # Copy right-side if done with left
        if l == left:
            merged[i] = a[left + r++]; continue
        # Copy left-side if done with right
        if r == right:
            merged[i] = a[l++]; continue

        # Copy the lesser of left or right side
        if a[l] <= a[left + r]
            merged[i] = a[l++]
        else
            merged[i] = a[left + r++]

    # Copy merged to original array
    copy(merged, a)
Merge Sort: Observations

Algorithm works **top-down** *(post-order traversal)*

End recursion when we have **1** element

If we split incorrectly (unevenly), then we get **insertion sort**

Merge requires **0(n)** extra space

Does not require **random access**
Merge Sort: Variants

1. **Hybrid**: switch to insertion sort for small N

2. Avoid **dynamic allocation** by using a **static** buffer

3. **Bottom-up** mergesort using a non-recursive **combine-and-conquer** strategy:
   
   a. View all elements of size 1.

   b. We scan through list performing 1-by-1 merges to produce ordered sublists of size 2

   c. Repeat for sizes 2, 4, 8, etc. until sorted (**level-order traversal**)
Merge Sort: Bottom Up

Continually merge adjacent subarrays of size 1, 2, 4, 8, etc.
Merge Sort: Properties

Best case: $O(n \log n)$

Worst case: $O(n \log n)$

Stable

Animations
Stable Sort
Stable Sorting

A sorting algorithm is considered **stable** if it maintains the **relative order** of **equivalent elements**.

For instance, given:

```
4 6¹ 6² 3 7
```

A **stable sort** would guarantee that 6¹ always occurs before 6² while an unstable sort could not guarantee this property.

Insertion Sort and Merge Sort are examples of **Stable Sorting** Algorithms.
Multi-dimensional Sorting

1. Use a **stable-sort** to sort multiple times

   Sort by **lowest priority** first

1. Use a **multi-factor comparison** function

   Sort by **highest priority** first
Multi-dimensional Sorting: Example

Given an array of containing students' first and last names, print out the students sorted by last name and then first name:

Using **stable sorting**

[Source Code](#)

Using **multi-factor comparison** function

[Source Code](#)