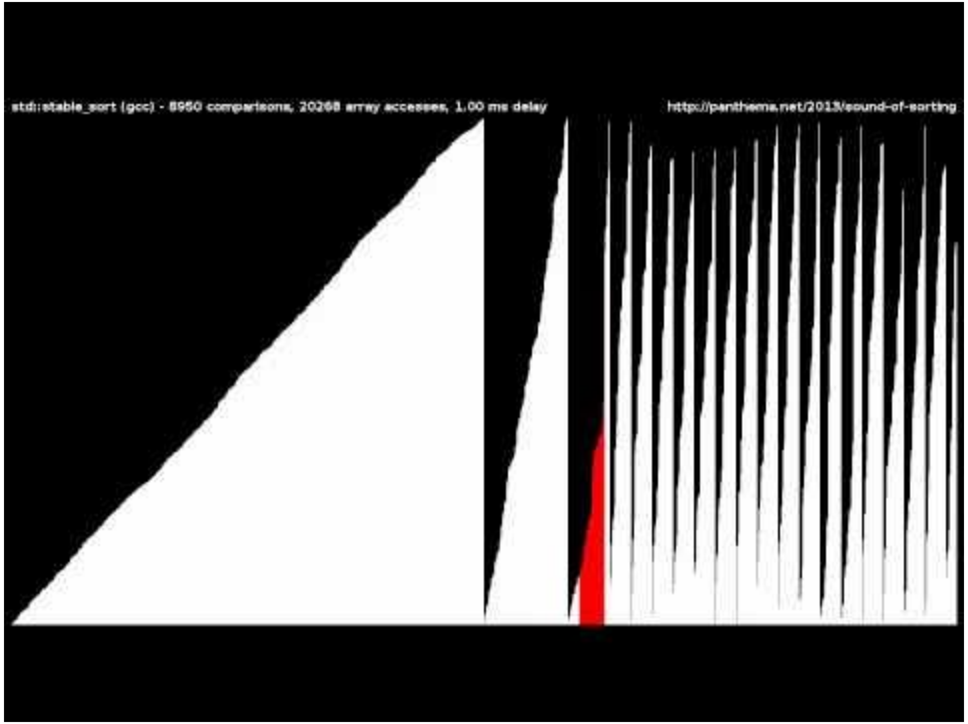
Lecture 12

Sorting II

Note: log n and log(n) are used interchangeably and both refer to $\log_2(n)$ unless otherwise specified.

- Sorting in O(n log(n))-time
    - Merge Sort
    - Quick Sort
    - Heap Sort
- Comparison Based Sorts
- Bucket Sort



Sorting Videos

Warning: Flashing Screen (especially for the first 8 seconds).

What I Did Last Summer
Friday Colloquium
2:30pm – 4pm
Retrogame Archeology

Retro Video Game Lab
Friday 11am – 12pm
Schow Library 014

entrance is near the bean bag chairs

Where will the log(n) come from?

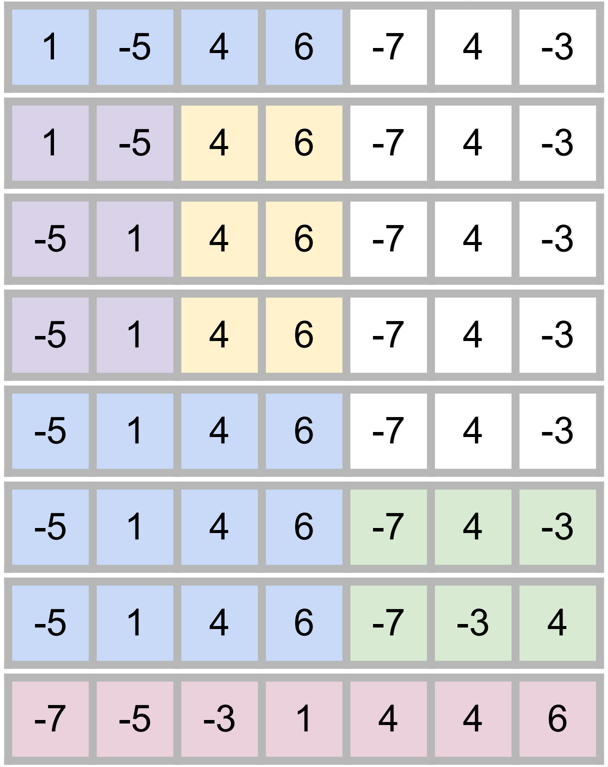
Sorting in O(n log(n))-time

Merge Sort

Merge Sort

Split the array into two halves. Sort each half recursively and merge them together.

- The base case is 1 element (or 2 elements shown below).

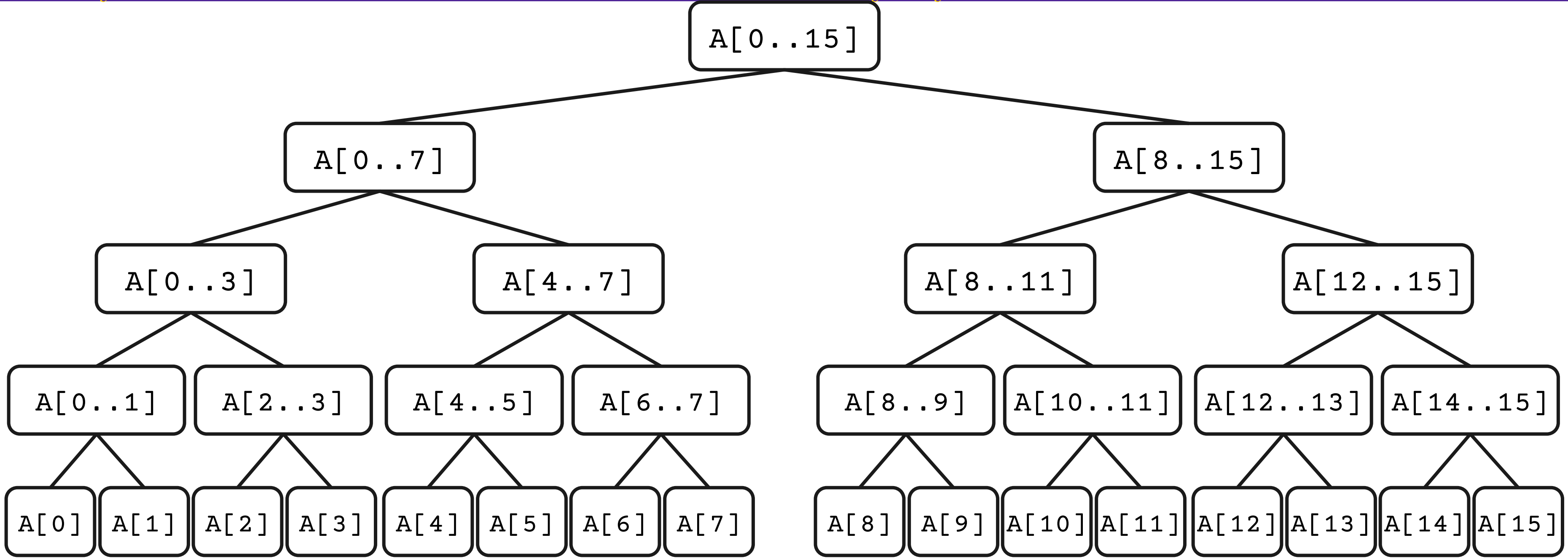


1	-5	4	6	-7	4	-3
1	-5	4	6	-7	4	-3
-5	1	4	6	-7	4	-3
-5	1	4	6	-7	4	-3
-5	1	4	6	-7	4	-3
-5	1	4	6	-7	4	-3
-5	1	4	6	-7	-3	4
-7	-5	-3	1	4	4	6

Some example steps of merge sort.

Analysis of Merge Sort

- It runs in O(n log n)-time in the worst-case.
  - There are log n levels in the recursion and each level takes a total of O(n)-time.
  - Illustration on the next slide.
- It is difficult to implement this in-place.
  - The merge step uses additional space.
- It is stable.
  - Why?



Merge sort on an array `A`, where each node is a recursive call with a range of indices to be sorted.

- We divide the range in half when going down, so the height of this tree is log(n).
- The merge step is O(k)-time where k is the number of values being merged.
  Therefore, we need to sum all of the ranges to get the overall run-time.
- Notice that each layer contains each element of A exactly once.
  So each layer requires a total of O(n)-time.

There are log(n) layers and each requires O(n)-time. Hence, the overall run-time is O(n log(n))-time.

Live Coding: `MergeSort`

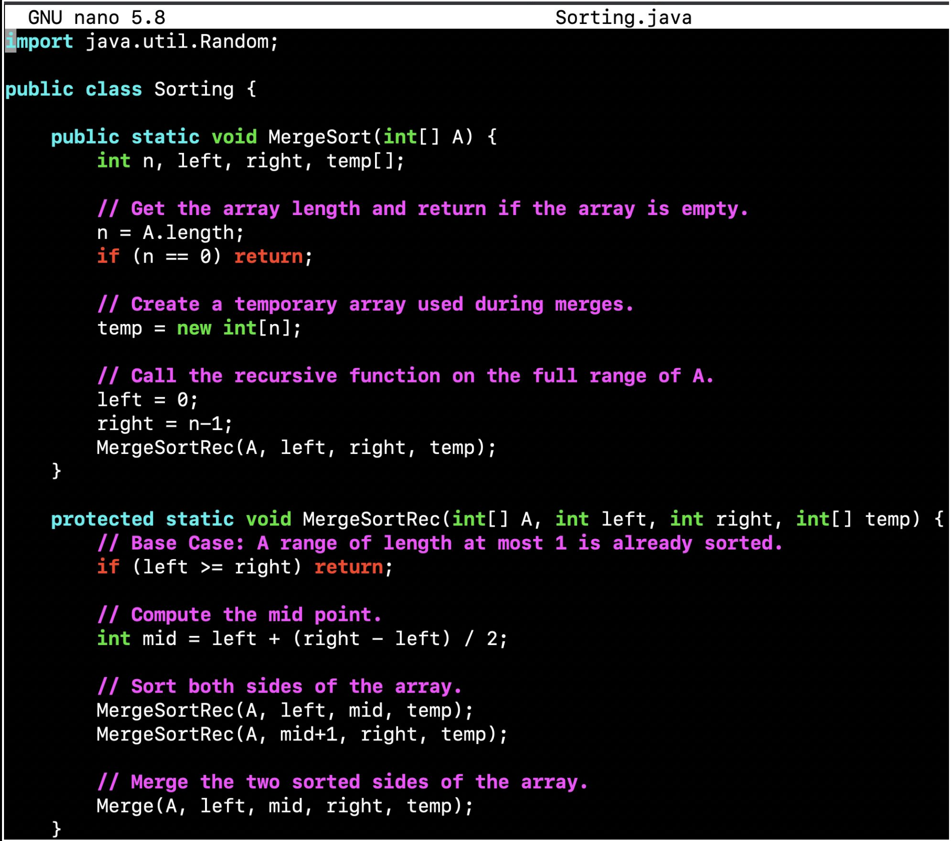
Let's implement merge sort!

- The function `MergeSort(int[] A)` modifies `A`.
  It will call the recursive function `MergeSortRec` that does all the work.
- The function `MergeSortRec` will use the following arguments:
  - `int A[]` is the array.
  - `int left` and `int right` specify that `A[left...right]` is to be sorted by this recursive call. Otherwise, we could copy subarrays during each recursive call, but this would be wasteful.
  - `int temp[]` will be used for temporary space during merging. It must have size at least `A.length()=n`. This temporary space will be created by the `MergeSort` function.
  - Therefore, the signature is `MergeSortRec(int[] A, int left, int right, int[] temp)`.

We also need to implement a helper function `merge`.

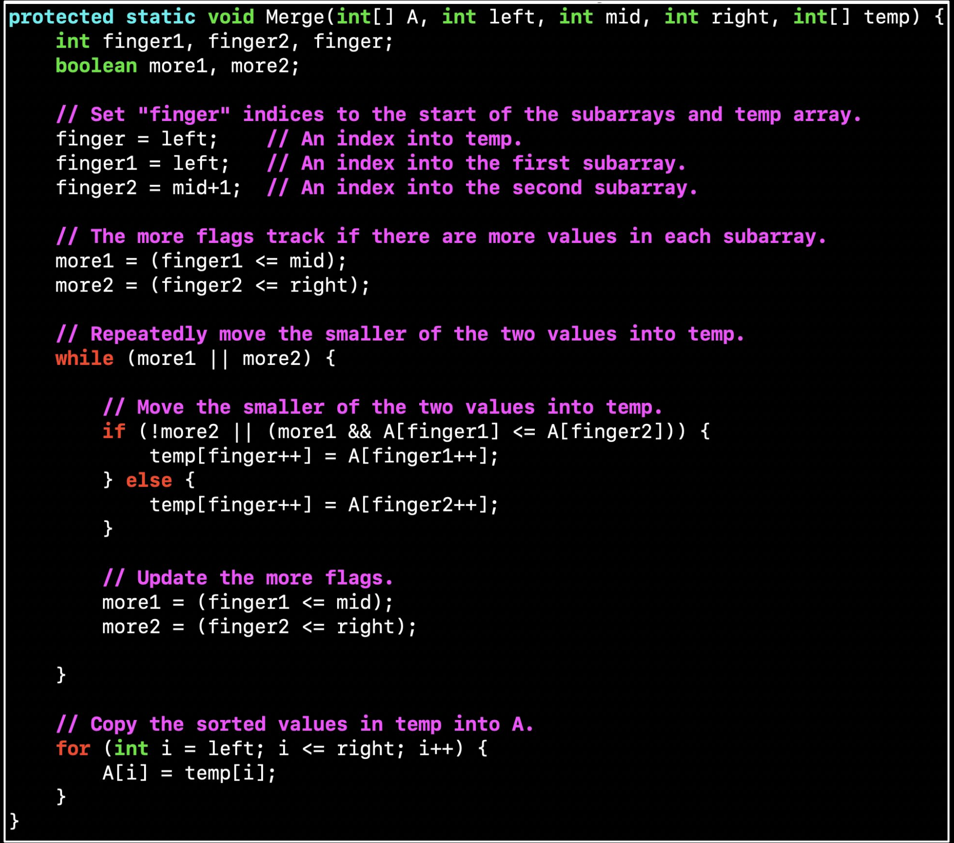
- Conceptually, its input is two sorted arrays, and it combines them into a single sorted array.
- In practice, merge sort on needs to merge two subarrays that are next to each other.
  - Thus, we can use `merge(int[] A, int first, int mid, int last, int[] temp)`.
- It will merge the values into the `temp` array and then copy these values back to `A`.

Note: There are many different ways to implement merge sort.



GNU nano 5.8 Sorting.java

```java
import java.util.Random;

public class Sorting {

    public static void MergeSort(int[] A) {
        int n, left, right, temp[];

        // Get the array length and return if the array is empty.
        n = A.length;
        if (n == 0) return;

        // Create a temporary array used during merges.
        temp = new int[n];

        // Call the recursive function on the full range of A.
        left = 0;
        right = n-1;
        MergeSortRec(A, left, right, temp);
    }

    protected static void MergeSortRec(int[] A, int left, int right, int[] temp) {
        // Base Case: A range of length at most 1 is already sorted.
        if (left >= right) return;

        // Compute the mid point.
        int mid = left + (right - left) / 2;

        // Sort both sides of the array.
        MergeSortRec(A, left, mid, temp);
        MergeSortRec(A, mid+1, right, temp);

        // Merge the two sorted sides of the array.
        Merge(A, left, mid, right, temp);
    }
```



```java
protected static void Merge(int[] A, int left, int mid, int right, int[] temp) {
    int finger1, finger2, finger;
    boolean more1, more2;

    // Set "finger" indices to the start of the subarrays and temp array.
    finger = left;    // An index into temp.
    finger1 = left;   // An index into the first subarray.
    finger2 = mid+1;  // An index into the second subarray.

    // The more flags track if there are more values in each subarray.
    more1 = (finger1 <= mid);
    more2 = (finger2 <= right);

    // Repeatedly move the smaller of the two values into temp.
    while (more1 || more2) {

        // Move the smaller of the two values into temp.
        if (!more2 || (more1 && A[finger1] <= A[finger2])) {
            temp[finger++] = A[finger1++];
        } else {
            temp[finger++] = A[finger2++];
        }

        // Update the more flags.
        more1 = (finger1 <= mid);
        more2 = (finger2 <= right);

    }

    // Copy the sorted values in temp into A.
    for (int i = left; i <= right; i++) {
        A[i] = temp[i];
    }
}
```

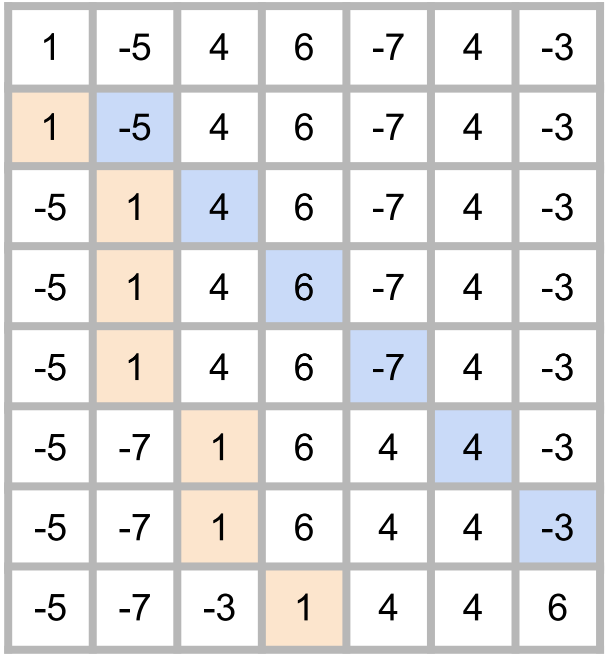
Implementing `MergeSort` (and `Merge`) in the file `Sorting.java`.

Quick Sort

Quicksort

Pick a random pivot and separate the items based on being smaller or larger.

- Use up to 2 swaps to separate each item and recursively sort the two subarrays.



1	-5	4	6	-7	4	-3
1	-5	4	6	-7	4	-3
-5	1	4	6	-7	4	-3
-5	1	4	6	-7	4	-3
-5	1	4	6	-7	4	-3
-5	-7	1	6	4	4	-3
-5	-7	1	6	4	4	-3
-5	-7	-3	1	4	4	6

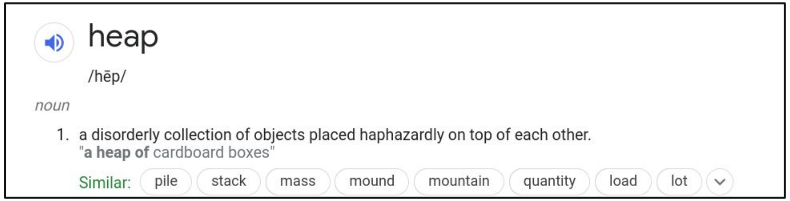
One pass of quicksort.
Notice that the last step swaps 6 with -3, and then -3 with 1.

<u>Analysis of Quicksort</u>

- It runs in O(n log n)-time in average cases.
    - Why?
- It is $O(n^2)$-time in the worst case.
    - When does this occur?
- One of the most efficient algorithms in practice.
- It is in-place.
- It is not stable.

Heap Sort

(preview)



heap

/hēp/

noun

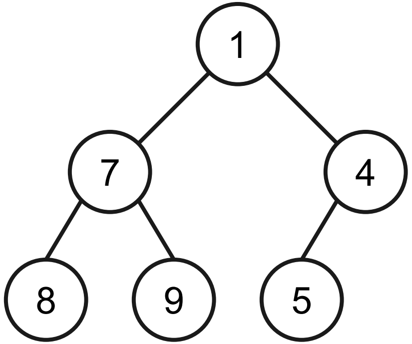
1. a disorderly collection of objects placed haphazardly on top of each other.
"**a heap of** cardboard boxes"

Similar: pile stack mass mound mountain quantity load lot

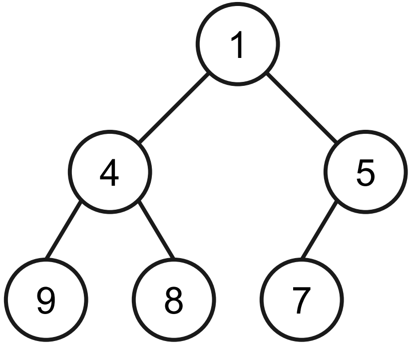
Binary Heaps

A *binary heap* is a binary tree whose node values satisfy two rules:

1. The value of a node is less than or equal to the values of its children.
2. The binary tree is *full* meaning that all levels except the bottom are full and the bottom level has all of the nodes as far to the left as is possible.



A binary heap.



Another binary heap with the same data.

We'll implement a binary heap with the following: where n is the number of nodes currently in the heap

(a) Insert a new value in log(n)-time. We preliminarily add it to the bottom and then fix the structure.

(b) Remove the minimum value in log(n)-time. Deleting the top element and fixing the structure.

How can we use these points as part of an O(n log(n))-time sorting algorithm?

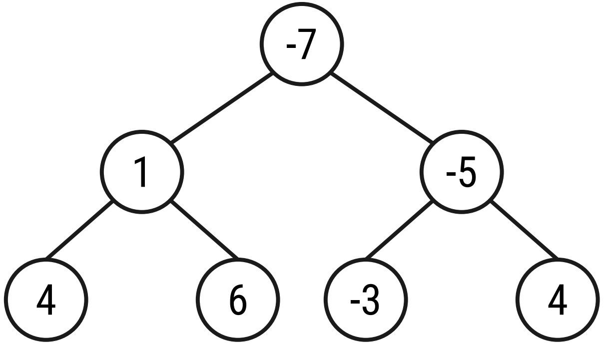
Heap Sort

Add all n items into a heap and then remove the minimum one at a time.

- The result is very similar to selection sort.
- More generally, any priority queue can be used to sort in this way.

Note: We’ll discuss binary trees, binary heaps, and priority queues later in the course.

1	-5	4	6	-7	4	-3



-7	-5	-3	1	4	4	6

Heap sort is based on the creation of a heap.

<u>Analysis of Heap Sort</u>

- It runs in O(n log n)-time in the worst-case.
  - Each insert takes O(log n)-time.
  - Each remove takes O(log n)-time.
- It is not in-place since we create a new array for the heap.
- It is not stable.

Comparison Based Sorting

Sorting Limitations

We have provide several algorithms for sorting in O(n log n)-time.
Is this the best possible result?

All of these algorithms have been *comparison-based* meaning that they only use the relative order of pairs of items via comparisons (<, ≤, =, ≥, >) to make decisions.

- The algorithms are focused on the *relative order* of the values, rather than any specific values.
  - For example, if you multiply every value by 10, then the algorithm works in exactly the same way.

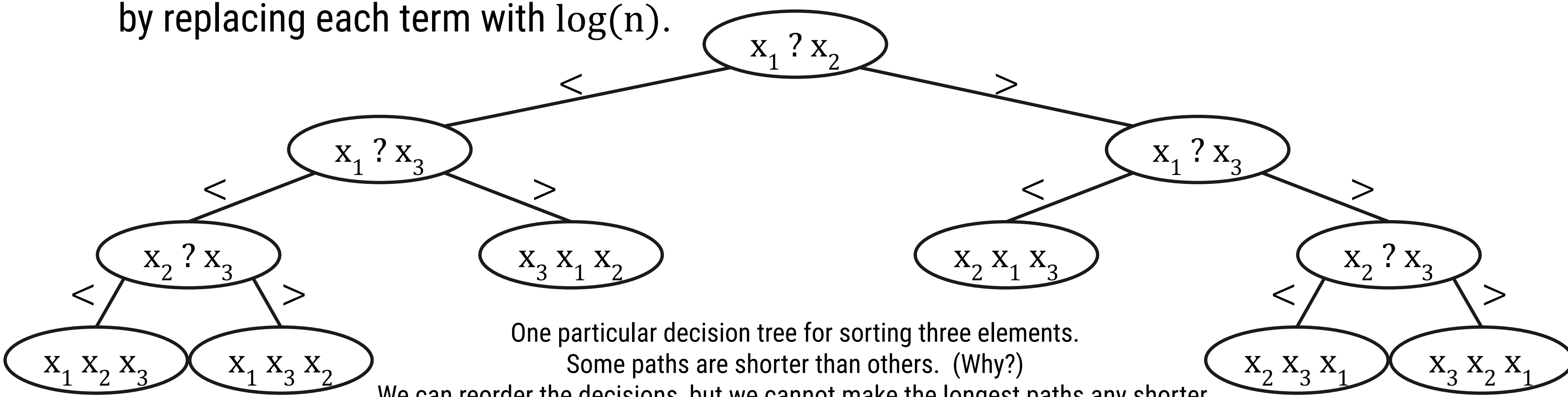
This leads to two questions:

1. Is O(n log n)-time the best possible for a comparison-based sorting algorithm?
2. Are there sorting algorithms that are not comparison-based?
   And could they run faster than O(n log n)-time?

Limitation of Comparison-Based Sorting Algorithms

A list of n distinct items has n! different permutations.

- Only one of the n! permutations is the correct sorted order. The algorithm must determine it.
- Each comparison x ? y divides the number of possibilities by two since x < y or x > y.
- $\log(n!) = \log(1 \cdot 2 \cdots n) = \log(1) + \log(2) + \cdots + \log(n) \leq \log(n) + \cdots + \log(n) = n \log(n)$ by replacing each term with $\log(n)$.



One particular decision tree for sorting three elements.
Some paths are shorter than others. (Why?)
We can reorder the decisions, but we cannot make the longest paths any shorter.

The last point implies that shortest binary tree with n! leaves has height O(n log n).

Theorem: Any comparison-based sort requires Ω(n log n)-time in the worst-case.

Bucket Sort

Bucket Sort

Suppose that we know that the minimum value is ≥ m and the maximum value is ≤ M.

- We may know these bounds in advance, or we can determine them exactly in O(n)-time.

Let's create an array of b = M - m + 1 "buckets" to hold each of the possible values.

1. Scan through the n values and put each value into its bucket. This takes O(n)-time.
   Note: Multiple items can go in the same bucket by using frequencies or an array of linked lists.
2. Scan through the buckets and put the values into a sorted array. This takes O(b+n)-time.

Buckets when the minimum is m=1 and the maximum is M=5.

An `int` in Java has values in the range -2147483648 to 2147483647, so b ≤ 2^{32} is a constant. Thus, bucket sorting `int` arrays is O(n)-time.

In practice, bucket sorting arbitrary `int` arrays can be slow and use gigabytes of storage (which *is* technically constant), but it is great for small b.

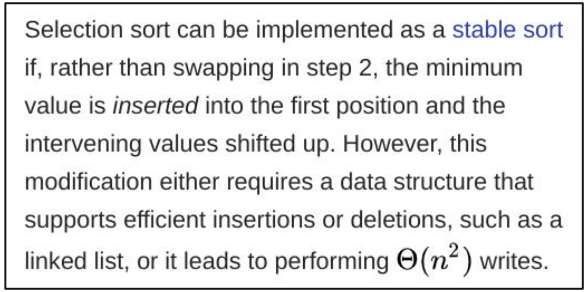
There are limitations to big-O analysis.

This algorithm does not use comparisons and it runs in O(n+b)-time.
Notice that O(n+b)-time is equal to O(n)-time whenever b ≤ c · n for some constant c.
In other words, if b is O(n) (i.e., the range is at most proportional to the number of values), then bucket sort is O(n)-time, which is faster than any comparison-based sorting algorithm.

Summary on Sorting

	Bubble	**Selection**	**Merge**	**Quick**	**Heap**	**Bucket**
comparison?	yes	yes	yes	yes	yes	no
worst-case	$O(n^2)$-time	$O(n^2)$-time	O(n log n)-time	$O(n^2)$-time	O(n log n)-time	O(n+b)-time
expected	$O(n^2)$-time	$O(n^2)$-time	O(n log n)-time	O(n log n)-time	O(n log n)-time	O(n+b)-time
in-place?	yes	yes	no	yes	no	no
stable?	yes	yes*	yes	no	no	yes

- Use library functions unless you have a special situation.
- Bucket sort is useful when we know there is a small range of possible values.
- Quicksort is usually the fastest in practice.
- Merge sort is an example of a divide-and-conquer algorithm.
- Heap sort is a data structure driven algorithm.
- Merge, Quick, Heap are asymptotically optimal in terms of total comparisons.
- Some of the above points require further explanation (e.g. Selection stability).



Selection sort can be implemented as a stable sort if, rather than swapping in step 2, the minimum value is *inserted* into the first position and the intervening values shifted up. However, this modification either requires a data structure that supports efficient insertions or deletions, such as a linked list, or it leads to performing $\Theta(n^2)$ writes.