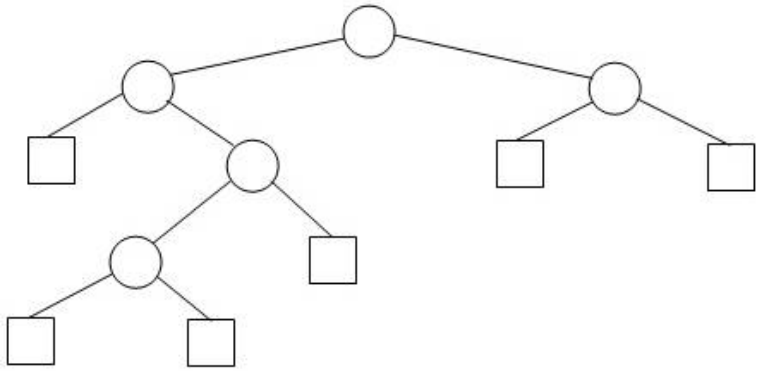
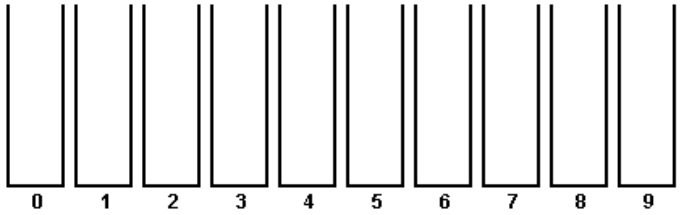


What is the min height of a tree with X external nodes?

# CSSE 230

## Sorting Lower Bound
## Radix Sort

Radix sort to the rescue … sort of…


After today, you should be able to…
…explain why comparison-based sorts need at least O(n log n) time
… explain bucket sort
… explain radix sort
… explain the situations in which radix sort is faster than O(n log n)

http://www.cs.auckland.ac.nz/software/AlgAnim/radixsort.html

# Announcements

- SortingRaces due Thursday
- The next class period after today will be in-class work day for Sorting Races

- The sounds of sorting. Radix sort later.
  - https://www.youtube.com/watch?v=kPRA0W1kECg

# A Lower-Bound on Sorting Time

We can't do much better than what we already know how to do.

# Insertion Sort

- Basic idea:
  - Think of the array as having a sorted part (at the beginning) and an unsorted part (the rest)

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| 38 | 44 | 87 | 2033 | 99 | 1500 | 100 | 90 | 239 | 748 |

  - Get the first value in the unsorted part
  - Insert it into the correct location in the sorted part, moving larger values up to make room

Repeat until unsorted part is empty

Q0

# Selection Sort: recursive version

```
static void selectionSortRec(int[] a) {
    selectionSortRec(a, a.length-1);
}

static void selectionSortRec(int[] a, int last) {
    if (last == 0) return;
    int largest = a[0];
    int largestPosition = 0;
    for (int j=1; j<=last; j++) {
        if (largest < a[j]) {
            largest = a[j];
            largestPosition = j;
        }
    }
    a[largestPosition] = a[last];
    a[last] = largest;
    selectionSortRec(a, last-1);
}
```

What's N?

Q0

# Insertion Sort: recursive version

```
// Recursive function to sort an array using
// insertion sort
void insertionSortRecursive(int arr[], int n)
{
    // Base case
    if (n <= 1)
        return;

    // Sort first n-1 elements
    insertionSortRecursive( arr, n-1 );

    // Insert last element at its correct position
    // in sorted array.
    int last = arr[n-1];
    int j = n-2;

    /* Move elements of arr[0..i-1], that are
      greater than key, to one position ahead
      of their current position */
    while (j >= 0 && arr[j] > last)
    {
        arr[j+1] = arr[j];
        j--;
    }
    arr[j+1] = last;
}
```

# Quick Sort & Merge Sort

```
void quicksort (int[] a, int lower, int upper)
{
    int pivotLocation = partition(a, lower, upper);
    quicksort(a,lower, pivotLocation - 1);
    quicksort(a,pivotLocation + 1, upper);
}

void mergeSort (int[] a) {
    int lower[a.length/2];
    int upper[a.length/2];
    split(a, lower, upper);
    mergeSort(lower);
    mergeSort(upper);
    merge(a, lower, upper);
}
```

# What's the best best case?

- Lower bound for best case?
- A particular algorithm that achieves this?

# What's the best worst case?

- Want a function f(N)
  such that the **worst case running time**
  for all sorting algorithms is Ω(f(N))

- How do we get a handle on
  "all sorting algorithms"?

Tricky!

- $\forall\, s: Sorting\ Algorithms\ (f_s(n)\ is\ \Omega(f(n))$
  - One counter example proves it false
  - Otherwise, to prove true must show all sorting algorithms are $\Omega(f(n))$

# What are “all sorting algorithms”?

- We can’t list all sorting algorithms and analyze all of them
  - Why not?

- But we can find a **uniform representation** of any sorting algorithm that is based on **comparing** elements of the array to each other

- Comparing means using: $\leq$

# First of all…

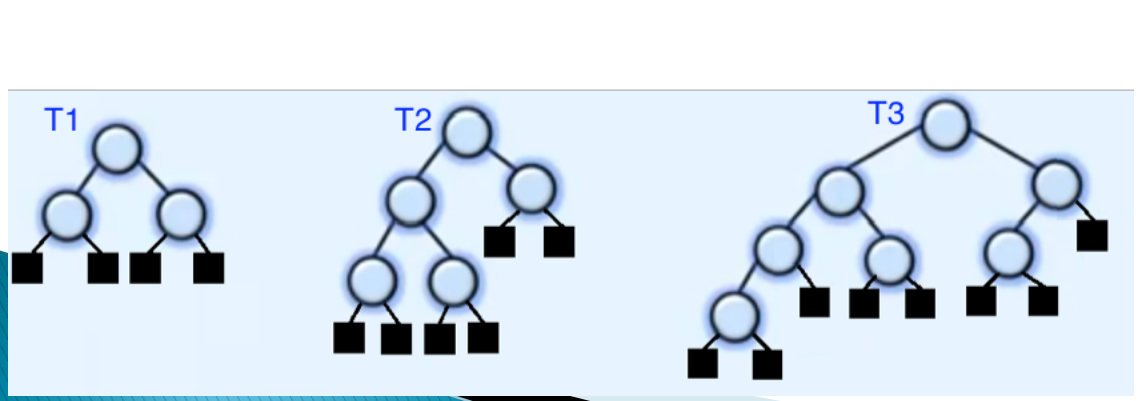
- The problem of sorting N elements is at least as hard as determining their ordering
  - e.g., determining that $a_3 < a_4 < a_1 < a_0 < a_2$

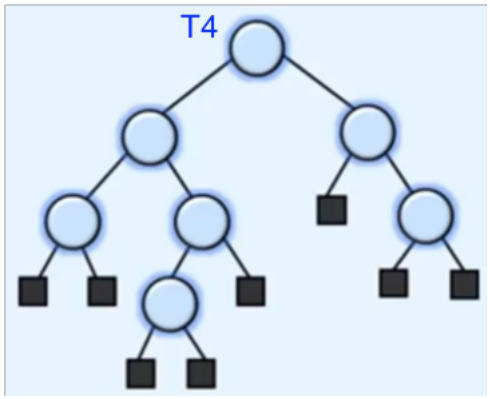
| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 58 | 55 | 73 | 5 | 10 |

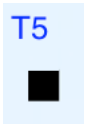
  - sorting = determining order, then movement
- So any lower bound on all "order-determination" algorithms is also a lower bound on "all sorting algorithms"

# A property of EBTs

- **Property** P(N): For any N ≥ 0, any EBT with N internal nodes has IN(T) + 1 external nodes.
- EN(T) = IN(T) + 1
- Use example EBTs T1 – T4 below to come up with the equation, let:
  - EN(T) = external nodes
  - IN(T) = internal nodes



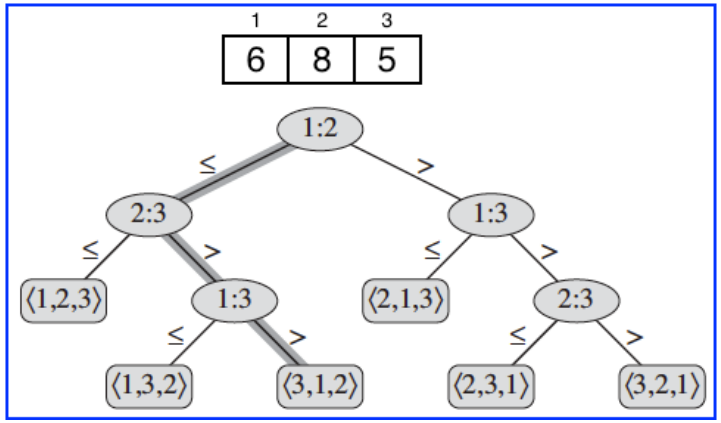



# Sort Decision Trees

- Let A be any **comparison-based algorithm** for sorting an array of distinct elements

- We can draw an EBT that corresponds to the comparisons that will be used by A to sort an array of N elements
  - This is called a **sort decision tree**
  - Internal nodes are comparisons
  - External nodes are orderings




  - Different algorithms will have different trees

Cormen 3e

***Theorem 8.1***

Any comparison sort algorithm requires $\Omega(n \lg n)$ comparisons in the worst case.

***Proof*** From the preceding discussion, it suffices to determine the height of a decision tree in which each permutation appears as a reachable leaf. Consider a decision tree of height $h$ with $l$ reachable leaves corresponding to a comparison sort on $n$ elements. Because each of the $n!$ permutations of the input appears as some leaf, we have $n! \leq l$. Since a binary tree of height $h$ has no more than $2^h$ leaves, we have

$$n! \leq l \leq 2^h ,$$

which, by taking logarithms, implies

$$\begin{aligned} h &\geq \lg(n!) && \text{(since the lg function is monotonically increasing)} \\ &= \Omega(n \lg n) && \text{(by equation (3.19))} . \end{aligned}$$

■

# So what?

- Minimum number of external nodes in a sort decision tree? (As a function of N)
- Is this number dependent on the algorithm?
- What's the height of the shortest EBT with that many external nodes?

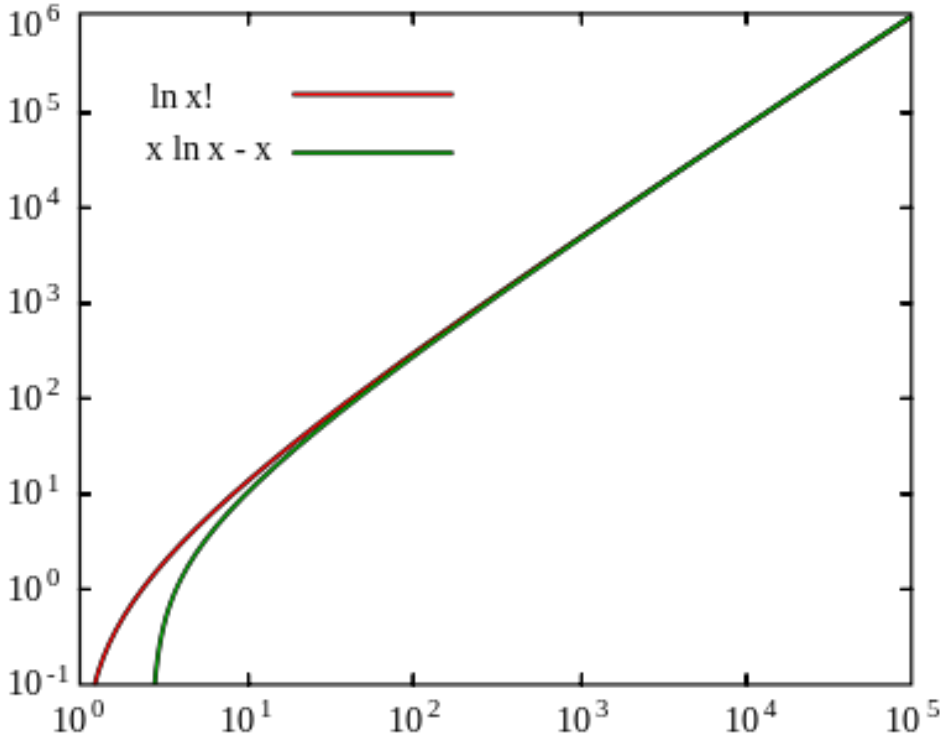
$$\lceil \log N! \rceil \approx N \log N - 1.44N = \Omega(N \log N)$$

No comparison-based sorting algorithm, known or not yet discovered, can **ever** do better than this!

# An approximation for log (n!)

- Use Stirling's approximation:

$$\ln n! = n \ln n - n + O(\ln(n))$$




http://en.wikipedia.org/wiki/Stirling%27s_approximation

# Can we do better than $N \log N$?

- $\Omega(N \log N)$ is the best we can do if we compare items
- Can we sort without comparing items?

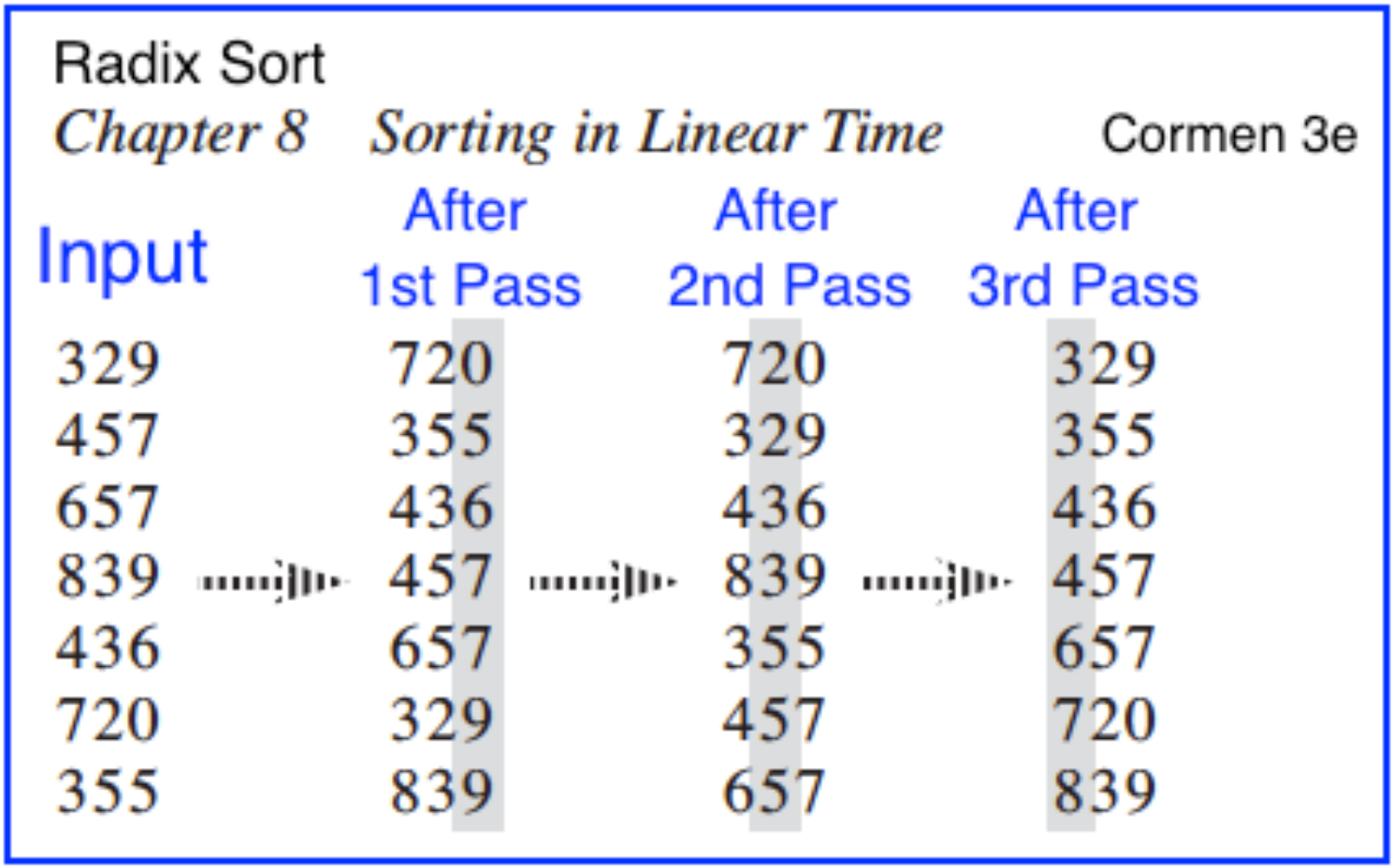
# Yes, we can! We can avoid comparing items and still sort. This is fast if the range of data is small. Q5

- Observation:
  - For N items, if the range of data is less than N, then we have duplicates
- O(N) sort: Bucket sort
  - Works if possible values come from limited range and have a uniform distribution over the range
  - Example: Exam grades histogram
- A variation: Radix sort

# Radix sort

- A picture is worth $10^3$ words, but an animation is worth $2^{10}$ pictures, so we will look at one.
- http://www.cs.auckland.ac.nz/software/AlgAnim/radixsort.html (good but blocked)
- https://www.youtube.com/watch?v=xuU-DS_5Z4g&src_vid=4S1L-pyQm7Y&feature=iv&annotation_id=annotation_133993417 (video, good basic idea, distracting zooms)
- http://www.cs.usfca.edu/~galles/visualization/RadixSort.html (good, uses single array)

# RadixSort is almost O(n)



# RadixSort is almost O(n)

- It is O(kn)
  - Looking back at the radix sort algorithm, what is k?
- Look at some extreme cases:
  - If all integers in range 0-99 (so, many duplicates if N is large), then k = _____
  - If all N integers are distinct, k = ____