Recall the sorting problem:
- input: a list \( L \) of \( n \) integers
- goal: return \( L \) sorted in increasing order

We've seen a variety of sorting algorithms:
- selection sort
- insertion sort
- merge sort
- heap sort
- quick sort
- radix sort

We've also known that any comparison-based sort is at best \( O(n \log n) \).

**Selection Sort**

**idea:** keep the \( p \) smallest elements at front, in order

Start with \( p = 0 \) (the smallest element may not be at the front)

- On each iteration, find the next smallest element (i.e., increment \( p \)).
- Swap with \( p \) (the smallest element may not be at the front).

Dec: keep the \( p \) smallest elements at front, in order.

**best-case running time:** \( O(n^2) \)

**worst-case running time:** \( O(n^2) \)

This is in-place because we only use the original space used by the input list.

and move it to the end of the sorted portion.

On each iteration, find the next smallest element (i.e., increment \( p \)).

Start with \( p = 0 \) (the smallest element may not be at the front).

We've reviewed and expanded

**CS 151**
**Insertion Sort**

Idea: Keep a growing "sorted list so far" in front portion of array.

Initially, the sorted portion contains only the first element (it's sorted).

On each iteration, take first element of unsorted portion, and insert it into the sorted portion at the correct position.

This is in-place because we make use of a new sorted list S on each call.

Best-case running time: \( O(n) \) if list is already sorted.

Worst-case running time: \( O(n^2) \)

**Mergesort**

Idea: If the given list has 0 or 1 elements, return it as sorted.

Otherwise, split the list into two equal-ish halves L and R.

Recursively sort L and R using mergesort.

Merge the sorted L and R:

- Initialize a new sorted list S to the empty list.
- Repeatedly remove the min remaining element from L and R.
- Merge the sorted L and R using mergesort.
- Return S.

This is not in-place because we make use of a new sorted list S on each call.

Best-case running time: \( O(n \log n) \) because always do some amount of work.

Worst-case running time: \( O(n \log n) \)

This is not in-place because we use only the original space used by the input list.

Best-case running time: \( O(n) \) if list is already sorted.

Worst-case running time: \( O(n^2) \)

**Heap Sort**

Idea: Insert each of the n elements into a heap.

Initialize a new sorted list S to the empty list.

While the heap is not empty:

- Remove the min remaining element from the heap, and add it to the end of our sorted list.

Return S.

This is not in-place because we make use of a new sorted list S on each call.

Best-case running time: \( O(n \log n) \) because always do some amount of work.

Worst-case running time: \( O(n \log n) \)

This is not in-place because we use only the original space used by the input list.

Best-case running time: \( O(n) \) if list is already sorted.

Worst-case running time: \( O(n \log n) \)

**Bucket Sort**

Idea: If we know the elements are all within the range 0...R for some int R.

Initialize an array A[0..R] (one bucket per range value).

For each element e, place it at A[e].

Initialize a new sorted list S to the empty list.

For i=0 to R, if A[i] is non-null, add A[i] to end of S.

Return S.

This is not in-place because we make use of a new sorted list S on each call.

Best-case running time: \( O(n) \) if list is already sorted.

Worst-case running time: \( O(n^2) \)
Radix sort

idea: if we know the elements are each a sequence of radii / bases
insert each element into a Trie T
initialize a new sorted list S to the empty list
loop through the Trie in order, adding isWord elements to end of S
return S

Best-case running time: O(?) b/c we have to insert and remove each elt.
Worst-case running time: O(?) where e* is the max-length element
This is not in-place because we make a possibly LARGE Trie T.
(And this Trie is O( #bases) in size, and likely << n.)

Quicksort

idea: if the given list has 0 or 1 elements, return it as sorted
pick any value v in list as a "pivot" value
partition list-v into L, R s.t. elts of L are < pivot, and elts of R are >= pivot
recursively quicksort L and R
return concatenation(L, pivot, R)

Best-case running time: O(n log n) b/c same RT as mergesort.
Worst-case running time: O(n^2) but AVERAGE is O(n log n)

RT: Let T(n) denote the running time of quicksort on a list of n elements.
Then T(n) = T(|L|) + T(|R|) + O(n) and T(0)=T(1)
∈ O(1)
When pivot is "bad", could have |L|=n-1 every time. If "good", |L| \in O(n)

Shellsort

idea: a multi-pass generalization of insertion sort
on each pass with some gap g, perform g insertion sorts on elts g apart
sequence of gaps is given to you, always ends with gap 1 (insertion sort)
idea is that successive sorts are more and more sorted... and insertion sort
works on almost-sorted lists
Best-case running time: O(n) for good gaps
Worst-case running time: O(n log n) for bad gaps

Lower Bound on Sorting

Theorem: every (comparison-based) algorithm for sorting is at least n log n.
Proof: Sorting can be thought of as determining one permutation of its input.
Theorem: every (comparison-based) algorithm for sorting is at least n log n.
Theorem: every (comparison-based) algorithm for sorting is at least \( n \log n \).

Proof: So, we start with \( n! \) possible orderings / permutations. We know that \( n! \) is two before we get down to 1. I.e. we need at least \( \log(n!) \) comparisons.

The number of comparisons is thus the number of times we can divide \( n! \) by 2 before we get down to one possibility. Thus, we need at least \( \log(n!) \) comparisons.

With each comparison, we divide the number of possibilities in two. If the algorithm ends when we are down to one possibility, then the number of comparisons is \( \log(n!) \).

Using Stirling's approximation (don't worry 'bout it), we know that \( \log(n!) \) is \( n \log n \).