InsertionSort

```java
// Code for sorting an array of int
for (int i = 1; i < a.length; i++) {
    int temp = a[i];
    int k = i;
    for (; 0 < k && temp < a[k-1]; k--)
        a[k] = a[k-1];
    a[k] = temp;
}
```

- Many people sort cards this way
- Invariant: everything to left of i is already sorted
- Works especially well when input is nearly sorted

**Runtime**
- Worst-case: O(n^2)
- Best-case: O(n)
- Expected-case: O(n^2)

Can count expected number of inversions
- For a sequence with n elements
- The average number of inversions is n(n-1)/4

SelectionSort

- To sort an array of size n:
  - Examine all elements from 0 to (n-1); find the smallest one and swap it with the 0th element of the array
  - Examine all elements from 1 to (n-1); find the smallest in that part of the array and swap it with the 1st element of the array
  - In general, at the ith step, examine array elements from i to (n-1); find the smallest element in that range, and exchange it with the ith element of the array

- This is the other common way for people to sort cards

**Runtime**
- Worst-case: O(n^2)
- Best-case: O(n^2)
- Expected-case: O(n^2)

Divide & Conquer?

- It often pays to
  1) break the problem into smaller subproblems,
  2) solve the subproblems separately, and then
  3) assemble a final solution

- This technique is called **Divide-and-Conquer**
- Caveat: the partitioning and assembly processes cannot be too expensive
- Can we apply this approach to sorting?

MergeSort

- Quintessential divide-and-conquer algorithm
- Divide array into equal parts, sort each part, then merge

**Runtime**
- Three questions:
  1. How do we divide array into two equal parts?
     - A1: Use indices into array
  2. How do we sort the parts?
     - A2: call MergeSort recursively!
  3. How do we merge the sorted subarrays?
     - A3: Have to write some (easy) code

Merging Sorted Arrays A and B

- Create an array C of size = size of A + size of B
- Keep three indices:
  - ai into A
  - bi into B
  - ci into C
- Initialize all three indices to 0 (start of each array)
- Compare element A[ai] with B[bi], and move the smaller element into C[ci]
- Increment the appropriate indices (ai or bi), and ci
- If either A or B is empty, copy remaining elements from the other array (B or A, respectively) into C
Merging Sorted Arrays

$C = \text{merged array}$

MergeSort Analysis

- Outline (text has detailed code)
  - Split array into two halves
  - Recursively sort each half
  - Merge the two halves

- Merge = combine two sorted arrays to make a single sorted array
  - Rule: Always choose the smallest item
  - Time: $O(n)$ where $n$ is the combined size of the two arrays

- Runtime recurrence
  - Let $T(n)$ be the time to sort an array of size $n$
  - $T(n) \leq 2T(n/2) + cn$
  - $T(1) = c$

- Can show by induction that $T(n) = O(n \log n)$
- Alternately, can show $T(n) = O(n \log n)$ by looking at tree of recursive calls

MergeSort Notes

- Asymptotic complexity: $O(n \log n)$
  - Much faster than $O(n^2)$
- Disadvantage
  - Need extra storage for temporary arrays
  - In practice, this can be a serious disadvantage, even though MergeSort is asymptotically optimal for sorting
  - Can do MergeSort in place, but this is very tricky (and it slows down the algorithm significantly)
- Are there good sorting algorithms that do not use so much extra storage?
  - Yes: QuickSort

QuickSort

- Intuitive idea
  - Given an array $A$ to sort, choose a pivot value $p$
  - Partition $A$ into two subarrays, $AX$ and $AY$
  - $AX$ contains only elements $\leq p$
  - $AY$ contains only elements $\geq p$
  - Sort subarrays $AX$ and $AY$ separately
  - Concatenate (not merge!) sorted $AX$ and $AY$ to produce sorted $A$
  - Note that concatenation is easier than merging

QuickSort Questions

- Key problems
  - How should we choose a pivot?
  - How do we partition an array in place?

- Choosing a pivot
  - Ideal pivot is median since this splits array in half
  - Unfortunately, computing the median is expensive
  - Popular heuristics
    - Use first value in array as pivot (this is a bad choice)
    - Use middle value in array as pivot
    - Use median of first, last, and middle values in array as pivot

- Partitioning in place
  - Can be done in $O(n)$ time
  - See next few slides
In-Place Partitioning

How can we move all the blues to the left of all the reds?

1. Keep two indices, LEFT and RIGHT
2. Initialize LEFT at start of array and RIGHT at end of array
3. Invariant: all elements to left of LEFT are blue
   all elements to right of RIGHT are red
4. Keep advancing indices until they pass, maintaining invariant

Now neither LEFT nor RIGHT can advance and maintain invariant.
We can swap red and blue pointed to by LEFT and RIGHT indices.
After swap, indices can continue to advance until next conflict.

QuickSort Analysis

- Runtime analysis (worst-case)
  - Partition can work badly producing this:
  - Runtime recurrence
    - $T(n) = T(n-1) + n$
  - This can be solved to show worst-case $T(n) = O(n^2)$

- Runtime analysis (expected-case)
  - More complex recurrence (see text)
  - Can solve to show expected $T(n) = O(n \log n)$
  - Can improve constant factor by avoiding QuickSort on small sets
    - Switch to InsertionSort (for example) for sets of size, say, 8 or less
  - Definition of small depends on language, machine, etc.

Sorting Algorithm Summary

- The ones we have discussed
  - Insertion Sort
  - Selection Sort
  - Merge Sort
  - Quick Sort
- Other sorting algorithms
  - Heap Sort (come back to this)
  - Shell Sort (in text)
  - Bubble Sort (nice name)
  - Radix Sort
  - Bin Sort
  - Counting Sort

Why so many? Do Computer Scientists have some kind of sorting fetish or what?
- Stable sorts: Ins, Sel, Mer
- Worst-case $O(n \log n)$: Mer, Hea
- Expected-case $O(n \log n)$: Mer, Hea, Qui
- Best for nearly-sorted sets: Ins
- No extra space needed: Ins, Sel, Hea
- Fastest in practice: Qui
- Least data movement: Sel

Lower Bounds on Sorting: Goals

- Goal: Determine the minimum time required to sort $n$ items
- Note: we want worst-case not best-case time
  - Best-case doesn’t tell us much; for example, we know InsertionSort takes $O(n)$ time on already-sorted input
  - We want to determine the worst-case time for the best-possible algorithm
- But how can we prove anything about the best possible algorithm?
  - Want to find characteristics that are common to all sorting algorithms
  - Let’s try looking at comparisons
Comparison Trees

• Any algorithm can be "unrolled" to show the comparisons that are (potentially) performed

Example

\[
\text{for }(i = 0; i < x.\text{length}; i++)
\]

\[
f(\text{if } x[i] < 0 \text{ then } x[i] = -x[i])\]

• In general, you get a comparison tree

• If the algorithm fails to terminate for some input then the comparison tree is infinite

• The height of the comparison tree represents the worst-case number of comparisons for that algorithm

Lower Bounds on Sorting: Notation

• Suppose we want to sort the items in the array B[]

• Let's name the items
  - \(a_i\) is the item initially residing in \(B[1]\), \(a_2\) is the item initially residing in \(B[2]\), etc.
  - In general, \(a_i\) is the item initially stored in \(B[i]\)

• Rule: an item keeps its name forever, but it can change its location
  - Example: after swap(B,1,5), \(a_1\) is stored in \(B[5]\) and \(a_5\) is stored in \(B[1]\)

The Answer to a Sorting Problem

• An answer for a sorting problem tells where each of the \(a_i\) resides when the algorithm finishes

• How many answers are possible?

• The correct answer depends on the actual values represented by each \(a_i\)

• Since we don’t know what the \(a_i\) are going to be, it has to be possible to produce each permutation of the \(a_i\)

• For a sorting algorithm to be valid it must be possible for that algorithm to give any of \(n!\) potential answers

Comparison Tree for Sorting

• Every sorting algorithm has a corresponding comparison tree

• The comparison tree must have \(n!\) (or more) leaves because a valid sorting algorithm must be able to get any of \(n!\) possible answers

Time vs. Height

• The worst-case time for a sorting method must be \(\geq\) the height of its comparison tree

• The height corresponds to the worst-case number of comparisons

• Each comparison takes \(\Theta(1)\) time

• The algorithm is doing more than just comparisons

• What is the minimum possible height for a binary tree with \(n!\) leaves?

  Height \(\geq \log(n!) = \Theta(n \log n)\)

• This implies that any comparison-based sorting algorithm must have a worst-case time of \(\Omega(n \log n)\)

  Note: this is a lower bound; thus, the use of big-Omega instead of big-O

Using the Lower Bound on Sorting

Claim: I have a PQ

• Insert time: \(O(1)\)

• GetMax time: \(O(1)\)

• True or false?

False (for general sets) because if such a PQ existed, it could be used to sort in time \(O(n)\)

Claim: I have a PQ

• Insert time: \(O(\log \log n)\)

• GetMax time: \(O(\log \log n)\)

• True or false?

False (for general sets) because it could be used to sort in time \(O(n \log \log n)\)

True for items with priorities in range \(1..n\) [van Emde Boas]

(Note: such a set can be sorted in \(O(n)\) time)
Sorting in Linear Time

There are several sorting methods that take linear time:

- **Counting Sort**
  - Sorts integers from a small range: \([0..k]\) where \(k = O(n)\)

- **Radix Sort**
  - The method used by the old card-sorters
  - Sorting time \(O(dn)\) where \(d\) is the number of “digits”

- How do these methods get around the \(\Omega(n \log n)\) lower bound?
  - They don’t use comparisons

- What sorting method works best?
  - QuickSort is best general-purpose sort (but it’s not stable)
  - Counting Sort or Radix Sort can be best for some kinds of data