ALGORITHMS AND DATA STRUCTURES

SORTING ALGORITHMS

THE PROBLEM OF SORTING

- The problem of sorting is a computational problem that can be solved using so-called *sorting algorithms*

THE PROBLEM OF SORTING

- The problem of sorting is a computational problem that can be solved using so-called *sorting algorithms*
- More formally, consider the problem of sorting stated as follows
  - **Input**: A sequence of n numbers $A = \{a_1, ..., a_n\}$
  - **Output**: A permutation (reordering) $A' = \{a'_1, ..., a'_n\}$ of the input sequence such that $a'_1 <= a'_2 <= ... <= a'_n$

TYPES OF SORTING ALGORITHMS

- A sorting algorithm is an algorithm that puts elements of a list in a certain order. The most frequently used orders are numerical order and lexicographical order

TYPES OF SORTING ALGORITHMS

- A sorting algorithm is an algorithm that puts elements of a list in a certain order. The most frequently used orders are numerical order and lexicographical order
- Sorting algorithms can be classified by:
  - *Computational complexity* (worst, average and best behavior) in terms of the size of the list (*N*)
  - *Computational complexity of swaps* (for "in-place" algorithms)

TYPES OF SORTING ALGORITHMS

- *Memory usage* and use of other computer resources
- *Stability*: stable sorting algorithms maintain the relative order of records with equal keys (i.e. values).
- Whether the algorithm is *serial* [this course] *or parallel*
- Among other criteria…

- Here we're going to only focus on serial algorithms. Although we will explore the other types of algorithms

(SLOWEST) SORTING ALGORITHMS

- Average-case performance **$O(N^p)$, $p > 2$**
  - Bogosort $O((N+1)!)$
  - Stooge sort $O(N^{2.7095...})$
- Not-so-worst-but-still-bad average-case performance **$O(N^2)$**
  - Selection sort $O(N^2)$
  - Bubble sort $O(N^2)$
  - Insertion sort $O(N^2)$

BOGOSORT

- Highly ineffective
- Use for pedagogical purposes mainly
- Two versions
  - deterministic: enumerates all permutations
  - randomized: randomly permutes its input

BOGOSORT

- Highly ineffective
- Use for pedagogical purposes mainly
- Two versions
  - deterministic: enumerates all permutations
  - randomized: randomly permutes its input
- Best-case performance: $O(N)$
- Worst-case performance: $O((N+1)!)$

BOGOSORT

- Pseudocode for bogosort:

```
FUNCTION bogoSort:
   INPUT: integer array A[n]
   OUTPUT: none
   USAGE: bogoSort(A)
BEGIN
   WHILE NOT sorted(A)
      shuffle(A)
   END
END // bogoSort
```

```
FUNCTION sorted:
   INPUT: integer array A[n]
   OUTPUT: boolean
   USAGE: flag = sorted(A)
BEGIN
   FOR i: 1, n−1
      IF A[i] > A[i+1] THEN
         RETURN FALSE
       END
   END
   RETURN TRUE
END // sorted
```

STOOGE SORT

- Recursive algorithm that goes like this…
  1. If key at start is larger than that at the end, swap them
  2. If there are 3 or more elements in the list, then:
     1. Stooge sort the initial 2/3 of the list
     2. Stooge sort the final 2/3 of the list
     3. Stooge sort the initial 2/3 of the list again

STOOGE SORT

- Recursive algorithm that goes like this…
  1. If key at start is larger than that at the end, swap them
  2. If there are 3 or more elements in the list, then:
     1. Stooge sort the initial 2/3 of the list
     2. Stooge sort the final 2/3 of the list
     3. Stooge sort the initial 2/3 of the list again
- Worst-case time complexity: **$O(N^{\lg(3)/\lg(3/2)})$**

STOOGE SORT

- Pseudocode for stooge sort:

```
FUNCTION stoogeSort:
   INPUT: integer array A[n], integer i, j
   OUTPUT: none
   USAGE: stoogeSort(A, i, j)
BEGIN
   IF A[i] > A[j] THEN
      swap(A[i], A[j])
   END
    IF j – i + 1 > 2 THEN
      t = (j – i + 1) / 3
      stoogeSort(A, i, j – t)
      stoogeSort(A, i + t, j)
      stoogeSort(A, i, j – t)
    END
END // stoogeSort
```

SELECTION SORT

- Simple and intuitive to understand
- Divides input into
  - Sorted array built from left to right
  - Remaining array to be sorted
- Algorithm finds smallest element and swaps it with leftmost unsorted key, moving unsorted array to the right
- Best-case and worst-case execution times: $O(N^2)$

SELECTION SORT

- Pseudocode for selection sort:

```
FUNCTION selectionSort:
   INPUT: integer array A[n]
   OUTPUT: none
   USAGE: selectionSort(A)
BEGIN
   FOR i: 1, n
      minIndex = i
      FOR j: i+1, n
         IF A[j] < A[minIndex] THEN
           minIndex = j
         END
      END
      swap(A[i], A[minIndex])
   END
END // selectionSort
```

BUBBLE SORT

- Compares adjacent elements and swaps them if necessary
- The array is read until no swaps are needed
- Large elements in array surface or "bubble" to the top
- Slow in practical problems compared to insertion sort
- Easily optimizable but not great improvement achieved
- Best-case and worst-case performance (simplest algorithm): ***$O(N^2)$***

BUBBLE SORT

- Pseudocode for bubble sort:

```
FUNCTION bubbleSort:
   INPUT: integer array A[n]
   OUTPUT: none
   USAGE: bubbleSort(A)
BEGIN
   FOR i: 1, n−1
      FOR j: 1, n−1
         IF A[j] > A[j+1] THEN
            swap(A[j], A[j+1])
         END
      END
   END
END // bubbleSort
```

BUBBLE SORT

- (Slightly better) pseudocode for bubble sort:

```
FUNCTION bubbleSort:
   INPUT: integer array A[n]
   OUTPUT: none
   USAGE: bubbleSort(A)
BEGIN
   swapped = TRUE
   WHILE swapped
      FOR j: 1, n-1
         IF A[j] > A[j+1] THEN
            swap(A[j], A[j+1])
            swapped = FALSE
         END
      END
   END
END // bubbleSort
```

There is an error in this pseudocode. Can you detect it?

INSERTION SORT

- Easy to implement, stable and online algorithm
- Generally fastest among slow (bubble and selection) sorting algorithms using comparisons
- Sorts array by sorting an increasingly larger subarray
- Efficient for quasi-sorted arrays
- Best-case time complexity: $O(N)$
- Wort-case and average-case time complexity: $O(N^2)$

INSERTION SORT

- Pseudocode for insertion sort:

```
FUNCTION insertionSort:
 INPUT: integer array A[n]
 OUTPUT: none
 USAGE: insertionSort(A)
BEGIN
 FOR i: 2, n
  b ← A[i]
  j ← i − 1
  WHILE j > 0 AND A[j] > b DO
   A[j+1] ← A[j]
   j ← j − 1
  END
  A[j+1] ← b
 END
END // insertionSort
```

INSERTION SORT

- Example:

Original	34	8	64	51	32	21	Pos.
After p = 1	8	34	64	51	32	21	1
After p = 2	8	34	64	51	32	21	0
After p = 3	8	34	51	64	32	21	1
After p = 4	8	32	34	51	64	21	3
After p = 5	8	21	32	34	51	64	4

BOUNDS FOR SIMPLE/SLOW SORTING

- We'll prove a couple of theorems that are useful when discussing simple or slow comparison sorts. For that, we need to define what an inversion is

BOUNDS FOR SIMPLE/SLOW SORTING

- We'll prove a couple of theorems that are useful when discussing simple or slow comparison sorts. For that, we need to define what an inversion is
    - **Inversion**: Let A be an array of *N* numbers labeled by indices running from 0 to *N*-1. An inversion, in the array A, is any ordered pair (i,j) having the property that i < j but A[i] > A[j]
- Example: (34,8), (34,32), (34,21), (64,51), (64,32), (64,21), (51,32), (51,21), and (32,21)

BOUNDS FOR SIMPLE/SLOW SORTING

- This is exactly the number of swaps needed to be performed by insertion sort
- Swapping two adjacent items removes exactly one inversion
- If the input is ordered, there's $O(N)$ work in the algorithm; hence, the running time of insertion sort is $O(N + I)$, where I is the number of inversions in the original array
- Thus insertion sort runs in linear time if the number of insertions is $O(N)$

BOUNDS FOR SIMPLE/SLOW SORTING

- Eliminating all inversions in an array implies that it is sorted

BOUNDS FOR SIMPLE/SLOW SORTING

- Eliminating all inversions in an array implies that it is sorted
- A sorted array does not have inversions

BOUNDS FOR SIMPLE/SLOW SORTING

- Eliminating all inversions in an array implies that it is sorted
- A sorted array does not have inversions
- Assumptions:
  1. No duplicate items
  2. Input is some permutation of the first N elements (only relative order is important)
  3. All elements in the array are equally likely

BOUNDS FOR SIMPLE/SLOW SORTING

- **Lemma**:
  The average number of inversions in an array of N distinct elements is $N(N-1)/4$

  How to prove it? What elements are needed?

BOUNDS FOR SIMPLE/SLOW SORTING

- **Lemma**:
  The average number of inversions in an array of N distinct elements is $N(N-1)/4$

  How to prove it? What elements are needed?

- Provides a very strong lower bound about any algorithm that only exchanges adjacent elements
- Implies that insertion sort is quadratic on average

BOUNDS FOR SIMPLE/SLOW SORTING

- **Theorem**:
  Any algorithm that sorts by exchanging adjacent elements requires $\Omega(N^2)$ time on average

  How to prove it? What elements are needed?

- Valid over an entire class of sorting algorithms, including those undiscovered, that perform only adjacent exchanges

- Including bubble sort, selection sort, and insertion sort

BOUNDS FOR SIMPLE/SLOW SORTING

- The lower bound shows that in order for a sorting algorithm to run in $o(N^2)$ time, it must do comparisons and, in particular, exchanges between elements that are far apart