Outline of Today’s Class
Design and Analysis of Algorithms for Sorting
Efficiency: Insertion Sort vs. Mergesort

Lecture: Analysis of Algorithms (CS483 - 001)
Amarda Shehu
Spring 2017
1 Outline of Today’s Class

2 Design and Analysis of Algorithms for Sorting
   • Case Study 1: Insertion Sort
   • Case Study 2: Mergesort

3 Efficiency: Insertion Sort vs. Mergesort
The Sorting Problem

- Problem: Sort real numbers in ascending order
- Problem Statement:
  - **Input**: A sequence of $n$ numbers $\langle a_1, \ldots, a_n \rangle$
  - **Output**: A permutation $\langle a'_1, \ldots, a'_n \rangle$ s.t. $a'_1 \leq a'_2 \leq \ldots \leq a'_n$
- There are many sorting algorithms. How many can you list?
An Incomplete List of Sorting Algorithms

- Selection sort
- Insertion sort
- Library sort
- Shell sort
- Gnome sort
- Bubble sort
- Comb sort
- Flash sort
- Bucket sort
- Radix sort
- Counting sort
- Pigeonhole sort
- Mergesort
- Quicksort
- Heap sort
- Smooth sort
- Binary tree sort
- Topological sort
Case Study 1: Insertion Sort

- Split in teams and recall the idea behind insertion sort

Hint:

![Card Image]
Case Study 1: Insertion Sort

- Split in teams and recall the idea behind insertion sort

Hint:

- If you ever sorted a deck of cards, you have done insertion sort
Case Study 1: Insertion Sort

- Split in teams and recall the idea behind insertion sort

Hint:

- If you ever sorted a deck of cards, you have done insertion sort
- Move over a card, insert it in correct position
Case Study 1: Insertion Sort

- Split in teams and recall the idea behind insertion sort

Hint:

- If you ever sorted a deck of cards, you have done insertion sort
- Move over a card, insert it in correct position
Case Study 1: Insertion Sort

- Split in teams and recall the idea behind insertion sort

**Hint:**

- If you ever sorted a deck of cards, you have done insertion sort
- Move over a card, insert it in correct position

- $j$ points to current element
- $1 \ldots j - 1$ are sorted deck of cards
- $j \ldots n$ is yet unsorted (pile)
- Termination: when $j > n$
Case Study 1: Insertion Sort

- Split in teams and recall the idea behind insertion sort

Hint:

- If you ever sorted a deck of cards, you have done insertion sort
- Move over a card, insert it in correct position

- \( j \) points to current element
- \( 1 \ldots j - 1 \) are sorted deck of cards
- \( j \ldots n \) is yet unsorted (pile)
- Basic operation: pick and insert \( A[j] \) correctly in \( A[1 \ldots j - 1] \)
- Termination: when \( j > n \)
Insertion Sort: Pseudocode and Trace

**InsertionSort** (array $A[1 \ldots n]$)

1: for $j \leftarrow 2$ to $n$ do
2: \hspace{1em} Temp $\leftarrow A[j]$
3: \hspace{1em} $i \leftarrow j - 1$
4: while $i > 0$ and $A[i] >$ Temp do
5: \hspace{2em} $A[i + 1] \leftarrow A[i]$
6: \hspace{1em} $i \leftarrow i - 1$
7: \hspace{1em} $A[i + 1] \leftarrow$ Temp

- Loop invariant: At the start of each iteration $j$, $A[1 \ldots j - 1]$ is sorted.
Insertion Sort: Formal Proof of Correctness

Initialization: At start of iteration $j = 2$, $A[1\ldots1]$ is sorted. Yes, invariant holds.
Insertion Sort: Formal Proof of Correctness

1. **Initialization:** At start of iteration $j = 2$, $A[1\ldots1]$ is sorted. Yes, invariant holds.

2. **Maintenance:** Supposing that after iteration $j$ the loop invariant holds, show that it still holds after the next iteration. Go over the pseudocode to convince yourselves of this.
Insertion Sort: Formal Proof of Correctness

1 **Initialization:** At start of iteration \( j = 2 \), \( A[1 \ldots 1] \) is sorted. Yes, invariant holds.

2 **Maintenance:** Supposing that after iteration \( j \) the loop invariant holds, show that it still holds after the next iteration. Go over the pseudocode to convince yourselves of this.

3 **Termination:** The algorithm terminates when \( j = n + 1 \). At this point, the loop invariant states that \( A[1 \ldots n] \) is sorted. That is, the entire sequence of elements is in sorted order.

Q. E. D
Insertion Sort: Formal Proof of Correctness

1. **Initialization:** At start of iteration $j = 2$, $A[1 \ldots 1]$ is sorted. Yes, invariant holds.

2. **Maintenance:** Supposing that after iteration $j$ the loop invariant holds, show that it still holds after the next iteration. Go over the pseudocode to convince yourselves of this.

3. **Termination:** The algorithm terminates when $j = n + 1$. At this point, the loop invariant states that $A[1 \ldots n]$ is sorted. That is, the entire sequence of elements is in sorted order.

Q. E. D

Note: the structure of the proof should remind you of proofs by induction. You are expected to work through formal proofs of correctness in this class.
Insertion Sort: Formal Proof of Correctness

1. **Initialization:** At start of iteration $j = 2$, $A[1\ldots1]$ is sorted. Yes, invariant holds.

2. **Maintenance:** Supposing that after iteration $j$ the loop invariant holds, show that it still holds after the next iteration. Go over the pseudocode to convince yourselves of this.

3. **Termination:** The algorithm terminates when $j = n + 1$. At this point, the loop invariant states that $A[1\ldots n]$ is sorted. That is, the entire sequence of elements is in sorted order.

Q. E. D

Note: the structure of the proof should remind you of proofs by induction. You are expected to work through formal proofs of correctness in this class.
Properties of Insertion Sort

- Insertion sort is stable. Why?
- Insertion sort is an in-place algorithm. Why?
Properties of Insertion Sort

- Insertion sort is stable. Why?
- Insertion sort is an in-place algorithm. Why?
  - What does this mean for the space requirements of the algorithm?
Properties of Insertion Sort

- Insertion sort is stable. Why?
- Insertion sort is an in-place algorithm. Why?
  - What does this mean for the space requirements of the algorithm?
- Insertion sort is an online algorithm. What does this mean?
Properties of Insertion Sort

- Insertion sort is stable. Why?
- Insertion sort is an in-place algorithm. Why?
  - What does this mean for the space requirements of the algorithm?
- Insertion sort is an online algorithm. What does this mean?
- Insertion sort implements the direct paradigm.
Properties of Insertion Sort

- Insertion sort is stable. Why?
- Insertion sort is an in-place algorithm. Why?
  - What does this mean for the space requirements of the algorithm?
- Insertion sort is an online algorithm. What does this mean?
- Insertion sort implements the direct paradigm.
- How efficient is insertion sort? Let’s analyze its running time.
Properties of Insertion Sort

- Insertion sort is stable. Why?
- Insertion sort is an in-place algorithm. Why?
  - What does this mean for the space requirements of the algorithm?
- Insertion sort is an online algorithm. What does this mean?
- Insertion sort implements the direct paradigm.
- How efficient is insertion sort? Let’s analyze its running time.
Let $T(n) =$time it takes InsertionSort to sort a sequence of $n$ elements. Let $c_i$ denote the constant time it takes to execute statement $S_i$ in line $i$. Start with $T(n) = \text{time}(S_1)$.

$$T(n) = \sum_{j=2}^{n} \{ c_1 + \text{time}(S_2) + \text{time}(S_3) + \text{time}(S_4) + \text{time}(S_7) \}$$
$$= \sum_{j=2}^{n} \{ c_1 + c_2 + c_3 + \text{time}(S_4) + \text{time}(S_7) \}$$
$$\leq \sum_{j=2}^{n} \{ c_1 + c_2 + c_3 + \sum_{i=0}^{j-1}(c_4 + c_5 + c_6) + c_7 \}$$
$$= (n - 1) \cdot (c_1 + c_2 + c_3 + c_7) + \sum_{j=2}^{n} \sum_{i=0}^{j-1}(c_4 + c_5 + c_6)$$
$$= (n - 1) \cdot (c_1 + c_2 + c_3 + c_7) + \sum_{j=2}^{n} j(c_4 + c_5 + c_6)$$
$$= (n - 1) \cdot (c_1 + c_2 + c_3 + c_7) + (c_4 + c_5 + c_6) \sum_{j=2}^{n} j$$
Insertion Sort: Running Time

\[
    = (n - 1) \cdot (c_1 + c_2 + c_3 + c_7) + (c_4 + c_5 + c_6) \cdot \left( \frac{n \cdot (n+1)}{2} - 1 \right) \\
= (n - 1)A + \left( \frac{n \cdot (n+1)}{2} - 1 \right)B
\]

So: \( T(n) \leq An - A + B \frac{n^2}{2} + B \frac{n}{2} - B \)

- What is \( T(n) \) in the best-case scenario?
- What is the worst-case scenario? What is \( T(n) \) in that case?
- What is the average running time \( T(n) \) of insertion sort?
Case Study 2: Mergesort

Basic Idea behind Mergesort:

- Mergesort implements the divide and conquer paradigm.
- Each execution divides the sequence of elements in two halves until single element subsequences remain.
- The sorted halves are then merged in a way that preserves the sorting order.

Mergesort\((\text{array} A, p, r)\)
1. if \( p < r \) then
2. \( q \leftarrow (p + r)/2 \)
3. Mergesort\((A, p, q)\)
4. Mergesort\((A, q + 1, r)\)
5. Merge\((A, p, q, r)\)

1. Trace Mergesort on the sequence \( \{5, 2, 4, 5, 6, 1\} \)
2. Prove correctness (hint: assume \( n = 2^k \) and follow the recursion to obtain a simple proof by induction)
Properties of Mergesort

- Mergesort is stable. Why?
- Mergesort is not an in-place algorithm. Why?
Properties of Mergesort

- Mergesort is stable. Why?
- Mergesort is not an in-place algorithm. Why?
  - What does this mean for the space requirements of the algorithm?
Properties of Mergesort

- Mergesort is stable. Why?
- Mergesort is not an in-place algorithm. Why?
  - What does this mean for the space requirements of the algorithm?
- Mergesort is not an online algorithm. What does this mean?
Properties of Mergesort

- Mergesort is stable. Why?
- Mergesort is not an in-place algorithm. Why?
  - What does this mean for the space requirements of the algorithm?
- Mergesort is not an online algorithm. What does this mean?
- Mergesort implements the divide and conquer paradigm.
Properties of Mergesort

- Mergesort is stable. Why?
- Mergesort is not an in-place algorithm. Why?
  - What does this mean for the space requirements of the algorithm?
- Mergesort is not an online algorithm. What does this mean?
- Mergesort implements the divide and conquer paradigm.
- How efficient is mergesort? Let's analyze its running time.
Properties of Mergesort

- Mergesort is stable. Why?
- Mergesort is not an in-place algorithm. Why?
  - What does this mean for the space requirements of the algorithm?
- Mergesort is not an online algorithm. What does this mean?
- Mergesort implements the divide and conquer paradigm.
- How efficient is mergesort? Let’s analyze its running time.
Mergesort: Running Time

Let $T(n)$ = time it takes Mergesort to sort a sequence of $n$ elements. Let $c$ denote the constant time it takes to sort a sequence of length $n = 1$.

$$T(n) = \begin{cases} 
  c & \text{if } n = 1 \\
  T(n/2) + T(n/2) + \text{time}(\text{Merge}(n/2, n/2)) & \text{if } n > 1 
\end{cases}$$

So:

$$T(n) = \begin{cases} 
  c & \text{if } n = 1 \\
  2T(n/2) + cn & \text{if } n > 1 
\end{cases}$$

where $cn$ is the time to merge two subsequences of length $n/2$. 

Comparing Insertion sort to Mergesort

- Which algorithm would you prefer and why?
- Which one is faster?
- What happens when you need in-place sorting?
- What about online sorting?
- What happens when the sequences are very long?
- How does Mergesort scale vs. Insertion sort?
Comparing Insertion sort to Mergesort

- Which algorithm would you prefer and why?
- Which one is faster?
- What happens when you need in-place sorting?
- What about online sorting?
- What happens when the sequences are very long?
- How does Mergesort scale vs. Insertion sort?
  - Need to develop notations to compare functions