Insertion Sort

https://cs.pomona.edu/classes/cs140/
Outline

Topics and Learning Objectives
• Specify an algorithm
• Prove correctness
• Analyze total running time

Exercises
• None
Extra Materials

• **Chapter 2** of *Introduction to Algorithms*, Third Edition

• [https://www.toptal.com/developers/sorting-algorithms/](https://www.toptal.com/developers/sorting-algorithms/)
Warm-Up

Sorting Problem
• Input : a sequence of numbers
• Output : a reordering of the input into nondecreasing order
• Assumptions: none

We want to
• Specify an algorithm
• Argue that it correctly sorts
• Analyze its running time
Insertion Sort

1. FUNCTION InsertionSort(array)
2.   FOR j IN [1..<array.length]
3.     key = array[j]
4.     i = j - 1
5.     WHILE i ≥ 0 && array[i] > key
6.       array[i + 1] = array[i]
7.       i = i - 1
8.     // Insert “key” into correct position to its left.
9.   RETURN array
1. FUNCTION InsertionSort(array)
2. FOR j IN [1..<array.length]
3. key = array[j]
4. i = j - 1
5. WHILE i ≥ 0 && array[i] > key
6. array[i + 1] = array[i]
7. i = i - 1 // Insert “key” into correct position to its left.
8. array[i + 1] = key
9. RETURN array
Insertion Sort – Proof of correctness

Lemma (loop invariant)

- At the start of the iteration with index $j$, the subarray $\text{array}[0..j-1]$ consists of the elements originally in $\text{array}[0..j-1]$, but in non-decreasing order.

What is a lemma?

an intermediate theorem in a proof

What is a theorem?

a proposition that can be proved by a chain of reasoning

1. FUNCTION InsertionSort(array)
2.  FOR $j$ IN $[1..<\text{array}.\text{length}]$
3.     key = $\text{array}[j]$
4.     $i = j - 1$
5.     WHILE $i \geq 0$ && $\text{array}[i] > key$
6.         $\text{array}[i + 1] = \text{array}[i]$
7.         $i = i - 1$
8.     ENDWHILE
9.     $\text{array}[i + 1] = key$
10.    RETURN $\text{array}$
Lemma (loop invariant)
• At the start of the iteration with index j, the subarray array[0 ..= j-1] consists of the elements originally in array[0 ..= j-1], but in non-decreasing order.

General conditions for loop invariants
1. Initialization: The loop invariant is satisfied at the beginning of the loop.
2. Maintenance: If the loop invariant is true before the ith iteration, then the loop invariant will be true before the i+1 iteration.
3. Termination: When the loop terminates, the invariant gives us a useful property that helps show that the algorithm is correct.

FUNCTION InsertionSort(array)
2. FOR j IN [1 ..< array.length]
3. key = array[j]
4. i = j - 1
5. WHILE i ≥ 0 && array[i] > key
6. array[i + 1] = array[i]
7. i = i - 1
8. array[i + 1] = key
9. RETURN array
Insertion Sort – Proof of correctness

1. **Initialization**: The loop invariant is satisfied at the beginning of the loop.

   Lemma (loop invariant)
   - At the start of the iteration with index j, the subarray `array[0 ..= j-1]` consists of the elements originally in `array[0 ..= j-1]`, but in non-decreasing order.
   - When j = 1, the subarray is `array[0 ..= 1-1]`, which includes only the first element of the array. The single element subarray is sorted.

```plaintext
1.FUNCTION InsertionSort(array)
2.  FOR j IN [1 ..< array.length]
3.    key = array[j]
4.    i = j - 1
5.    WHILE i ≥ 0 && array[i] > key
6.      array[i + 1] = array[i]
7.      i = i - 1
8.  array[i + 1] = key
9. RETURN array
```
Insertion Sort – Proof of correctness

2. **Maintenance**: If the loop invariant is true before the $i$th iteration, then the loop invariant will be true before the $i+1$ iteration.

Lemma (loop invariant)

- At the start of the iteration with index $j$, the subarray $\text{array}[0 ..= j-1]$ consists of the elements originally in $\text{array}[0 ..= j-1]$, but in non-decreasing order.

- Assume $\text{array}[0 ..= j-1]$ is sorted. Informally, the loop operates by moving elements to the right until it finds the position of key. Next, $j$ is incremented.
for (i = 0; i < n; i++)

3

i = 0
while i < n

3

i++
Insertion Sort – Proof of correctness

3. **Termination**: When the loop terminates, the invariant gives us a useful property that helps show that the algorithm is correct.

Lemma (loop invariant)
- At the start of the iteration with index $j$, the subarray $\text{array}[0..j-1]$ consists of the elements originally in $\text{array}[0..j-1]$, but in non-decreasing order.
- The loop terminates when $j = n$. Given the initialization and maintenance results, we have shown that: $\text{array}[0..j-1] \rightarrow \text{array}[0..n-1]$ in non-decreasing order.
Insertion Sort – Running time

Analyze using the **RAM** (random access machine) model

- Instructions are executed one after another (no parallelism)
- Each instruction takes a constant amount of time
  - Arithmetic (+, -, *, /, %, floor, ceiling)
  - Data movement (load, store, copy)
  - Control (branching, subroutine calls)
- **Ignores memory hierarchy!** *(never forget: linked lists are awful)*
- This is a very simplified way of looking at algorithms
- Compare algorithms while ignoring hardware
Insertion Sort – Running time

On what does the running time depend?

• Number of items to sort (example, 1000 vs 3 numbers)

```
1. FUNCTION InsertionSort(array)
2. FOR j IN [1 ..< array.length]
3.     key = array[j]
4.     i = j - 1
5.     WHILE i ≥ 0 && array[i] > key
6.         array[i + 1] = array[i]
7.         i = i - 1
8.     array[i + 1] = key
9. RETURN array
```
Insertion Sort – Running time

On what does the running time depend?

- Number of items to sort (example, 1000 vs 3 numbers)
- How much are they already sorted
  - The hint here is that the inner loop is a **while** loop (not a for loop)
1. **FUNCTION** `InsertionSort(array)`
2. **FOR** `j` **IN** `[1 ..< array.length]`
3.     `key = array[j]`
4.     `i = j - 1`
5.     **WHILE** `i ≥ 0` **AND** `array[i] > key`
6.         `array[i + 1] = array[i]`
7.         `i = i - 1`
8.     `array[i + 1] = key`
9. **RETURN** `array`

<table>
<thead>
<tr>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
</tr>
<tr>
<td>2.</td>
</tr>
</tbody>
</table>
1. **FUNCTION** InsertionSort(array)  
2. \( j = 1 \)  
3. **WHILE** \( j < \text{array.length} \)  
4. \( \text{key} = \text{array}[j] \)  
5. \( i = j - 1 \)  
6. **WHILE** \( i \geq 0 \) && \( \text{array}[i] > \text{key} \)  
7. \( \text{array}[i + 1] = \text{array}[i] \)  
8. \( i = i - 1 \)  
9. \( \text{array}[i + 1] = \text{key} \)  
10. \( j = j + 1 \)  
11. **RETURN** array

<table>
<thead>
<tr>
<th>Cost</th>
</tr>
</thead>
</table>
| 1.   | 0  
| 2.   | 1  
| 3.   | 2  
| 4.   | 2  
| 5.   | 2  
| 6.   | 4  
| 7.   | 4  
| 8.   | 2  
| 9.   | 3  
| 10.  | 2  
| 11.  | 1  

### insertionSort Function

1. **FUNCTION** `insertionSort` *(array)*
2. \( j = 1 \)
3. **WHILE** \( j < array\.length \)
4. \( \text{key} = array[j] \)
5. \( i = j - 1 \)
6. **WHILE** \( i \geq 0 \ \&\& \ array[i] > \text{key} \)
7. \( array[i + 1] = array[i] \)
8. \( i = i - 1 \)
9. \( array[i + 1] = \text{key} \)
10. \( j = j + 1 \)
11. **RETURN** `array`

### Cost and Executions Table

<table>
<thead>
<tr>
<th></th>
<th>Cost</th>
<th>Executions</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2.</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3.</td>
<td>2</td>
<td>length</td>
</tr>
<tr>
<td>4.</td>
<td>2</td>
<td>length</td>
</tr>
<tr>
<td>5.</td>
<td>2</td>
<td>length</td>
</tr>
<tr>
<td>6.</td>
<td>4</td>
<td>length</td>
</tr>
<tr>
<td>7.</td>
<td>4</td>
<td>length</td>
</tr>
<tr>
<td>8.</td>
<td>2</td>
<td>length</td>
</tr>
<tr>
<td>9.</td>
<td>3</td>
<td>length</td>
</tr>
<tr>
<td>10.</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>11.</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
1. **FUNCTION** InsertionSort(array)

2. \( j = 1 \)

3. **WHILE** \( j < \) array.length

4. \( \text{key} = \) array[j]

5. \( i = j - 1 \)

6. **WHILE** \( i \geq 0 \) \&\& array[i] > key

7. \( \text{array}[i + 1] = \text{array}[i] \)

8. \( i = i - 1 \)

9. \( \text{array}[i + 1] = \text{key} \)

10. \( j = j + 1 \)

11. **RETURN** array

<table>
<thead>
<tr>
<th>Cost</th>
<th>Executions</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. 0</td>
<td>0</td>
</tr>
<tr>
<td>2. 1</td>
<td>1</td>
</tr>
<tr>
<td>3. 2</td>
<td>n</td>
</tr>
<tr>
<td>4. 2</td>
<td>n - 1</td>
</tr>
<tr>
<td>5. 2</td>
<td>n - 1</td>
</tr>
<tr>
<td>6. 4</td>
<td>?</td>
</tr>
<tr>
<td>7. 4</td>
<td></td>
</tr>
<tr>
<td>8. 2</td>
<td></td>
</tr>
<tr>
<td>9. 3</td>
<td></td>
</tr>
<tr>
<td>10. 2</td>
<td></td>
</tr>
<tr>
<td>11. 1</td>
<td></td>
</tr>
</tbody>
</table>

Loop code always executes one fewer time than the condition check.
1. **FUNCTION** InsertionSort(array)

2. \( j = 1 \)

3. **WHILE** \( j < \text{array}.\text{length} \)

4. \( \text{key} = \text{array}[j] \)

5. \( i = j - 1 \)

6. **WHILE** \( i \geq 0 \) \&\& \( \text{array}[i] > \text{key} \)

7. \( \text{array}[i + 1] = \text{array}[i] \)

8. \( i = i - 1 \)

9. \( \text{array}[i + 1] = \text{key} \)

10. \( j = j + 1 \)

11. **RETURN** array

---

<table>
<thead>
<tr>
<th>Cost</th>
<th>Executions</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. ( 0 )</td>
<td>1. ( 0 )</td>
</tr>
<tr>
<td>2. ( 1 )</td>
<td>2. ( 1 )</td>
</tr>
<tr>
<td>3. ( 2 )</td>
<td>( n ) ( n - 1 )</td>
</tr>
<tr>
<td>4. ( 2 )</td>
<td>( n - 1 )</td>
</tr>
<tr>
<td>5. ( 2 )</td>
<td>( n - 1 )</td>
</tr>
<tr>
<td>6. ( 4 )</td>
<td>( \text{depends} )</td>
</tr>
</tbody>
</table>

---

Loop code always executes one fewer time than the condition check.

Depends on how sorted array is
1. **FUNCTION** InsertionSort(array)

2. \( j = 1 \)

3. **WHILE** \( j < \) array.length

4. \( \text{key} = \text{array}[j] \)

5. \( i = j - 1 \)

6. **WHILE** \( i \geq 0 \) && \( \text{array}[i] > \text{key} \)

7. \( \text{array}[i + 1] = \text{array}[i] \)

8. \( i = i - 1 \)

9. \( \text{array}[i + 1] = \text{key} \)

10. \( j = j + 1 \)

11. **RETURN** array

<table>
<thead>
<tr>
<th></th>
<th>Cost</th>
<th>Executions</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>n</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>n - 1</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>n - 1</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>(n - 1)x</td>
</tr>
<tr>
<td>7</td>
<td>4</td>
<td>(n - 1)(x - 1)</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
<td>(n - 1)(x - 1)</td>
</tr>
<tr>
<td>9</td>
<td>3</td>
<td>n - 1</td>
</tr>
<tr>
<td>10</td>
<td>2</td>
<td>n - 1</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Loop code always execute one fewer time than the condition check.

Depends on how sorted array is
What is the total running time (add up all operations)?
“Talk To Your Neighbors”

- Spend 30 seconds working on the problem silently and alone.
- I will then tell you to work with your neighbors
- Finally, I will ask for an answer
- If I don’t get an answer, I will pick someone at random
**FUNCTION** InsertionSort(array)

1. **function** InsertionSort(array)
2. j = 1
3. **while** j < array.length
4. key = array[j]
5. i = j - 1
6. **while** i ≥ 0 && array[i] > key
7. array[i + 1] = array[i]
8. i = i - 1
9. array[i + 1] = key
10. j = j + 1
11. RETURN array

<table>
<thead>
<tr>
<th>Cost</th>
<th>Executions</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>0</td>
</tr>
<tr>
<td>2.</td>
<td>1</td>
</tr>
<tr>
<td>3.</td>
<td>2</td>
</tr>
<tr>
<td>4.</td>
<td>2</td>
</tr>
<tr>
<td>5.</td>
<td>2</td>
</tr>
<tr>
<td>6.</td>
<td>4</td>
</tr>
<tr>
<td>7.</td>
<td>4</td>
</tr>
<tr>
<td>8.</td>
<td>2</td>
</tr>
<tr>
<td>9.</td>
<td>3</td>
</tr>
<tr>
<td>10.</td>
<td>2</td>
</tr>
<tr>
<td>11.</td>
<td>1</td>
</tr>
</tbody>
</table>

What is the total running time (add up all operations)?

Total Running Time = 1 + 2n + (n - 1)(2 + 2 + 4x + (x - 1)(4 + 2) + 3 + 2) + 1
= 10nx + 5n - 10x - 1

Loop code always execute one fewer time than the condition check.

Depends on how sorted array is
**1. FUNCTION InsertionSort(array)**

2. \( j = 1 \)

3. **WHILE** \( j < \) array.length

4. \( \text{key} = \text{array}[j] \)

5. \( i = j - 1 \)

6. **WHILE** \( i \geq 0 \) \&\& \( \text{array}[i] > \text{key} \)

7. \( \text{array}[i + 1] = \text{array}[i] \)

8. \( i = i - 1 \)

9. \( \text{array}[i + 1] = \text{key} \)

10. \( j = j + 1 \)

11. **RETURN** array

<table>
<thead>
<tr>
<th></th>
<th>Cost</th>
<th>Executions</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2.</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3.</td>
<td>2</td>
<td>n</td>
</tr>
<tr>
<td>4.</td>
<td>2</td>
<td>( n - 1 )</td>
</tr>
<tr>
<td>5.</td>
<td>2</td>
<td>( n - 1 )</td>
</tr>
<tr>
<td>6.</td>
<td>4</td>
<td>( (n - 1)x )</td>
</tr>
<tr>
<td>7.</td>
<td>4</td>
<td>( (n - 1)(x - 1) )</td>
</tr>
<tr>
<td>8.</td>
<td>2</td>
<td>( (n - 1)(x - 1) )</td>
</tr>
<tr>
<td>9.</td>
<td>3</td>
<td>( n - 1 )</td>
</tr>
<tr>
<td>10.</td>
<td>2</td>
<td>( n - 1 )</td>
</tr>
<tr>
<td>11.</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Loop code always execute one fewer time than the condition check.

Depends on how sorted array is.

Is "- 11" a problem? Negative time?

**What is the best-case scenario?**

**array is already sorted**

\[ x = \]
FUNCTION InsertionSort(array)

1. j = 1
2. WHILE j < array.length
3. key = array[j]
4. i = j - 1
5. WHILE i ≥ 0 && array[i] > key
6. array[i + 1] = array[i]
7. i = i - 1
8. array[i + 1] = key
9. j = j + 1
10. RETURN array

Cost | Executions
-----|----------
1. 0  | 0        
2. 1  | 1        
3. 2  | n        
4. 2  | n - 1    
5. 2  | n - 1    
6. 4  | (n - 1)x 
7. 4  | (n - 1)(x - 1) 
8. 2  | (n - 1)(x - 1) 
9. 3  | n - 1    
10. 2 | n - 1    
11. 1 | 1        

What is the worst-case scenario? array is reverse sorted

Total Running Time = 1 + 2n + (n - 1)(2 + 2 + 4x + (x - 1)(4 + 2) + 3 + 2) + 1
= 10nx + 5n - 10x - 1
= 5n^2 + 5n - 5n - 1
= 5n^2 - 1

Loop code always execute one fewer time than the condition check.

Depends on how sorted array is

x = n/2 on average
Best, Worst, and Average

We usually concentrate on worst-case
• Gives an upper bound on the running time for any input
• The worst case can occur fairly often
• The average case is often relatively as bad as the worst case