Overview

- Review: Insertion Sort
- Merge Sort
- Quicksort
- Heapsort
- Counting Sort

For further reading, refer to Open Data Structures Book (Chapter 11)
Sorting

- **Input:**
  - a sequence of $n$ objects: $A[0], \ldots, A[n-1]$ (typically an array or a linked list)
  - a comparison predicate, $\leq$, that defines a total order on $A$

- **Output:**
  - an ordered representation of the objects in $A$
Sorting

**Input:**
- a sequence of $n$ objects: $A[0], \ldots, A[n − 1]$
  (typically an array or a linked list)
- a comparison predicate, $\leq$, that defines a total order on $A$

**Output:**
- an ordered representation of the objects in $A$

Many sorting algorithms exist:
- bubble sort, insertion sort, merge sort, heapsort, radix sort, bucket sort, quicksort, etc.
Insertion Sort

- Go through the items in the array (list) one by one
- For each item $x$ at index $i$:
  - We know the sub-array $A[0] \ldots A[i-1]$ is sorted
  - **Insert** $x$ in its correct position in the sub-array $A[i] \ldots A[i]$. 

| 4 | 3 | 2 | 10 | 12 | 1 | 5 | 6 |
Insertion Sort

- Go through the items in the array (list) one by one
- For each item $x$ at index $i$:
  - We know the sub-array $A[0] \ldots A[i - 1]$ is sorted
  - **Insert** $x$ in its correct position in the sub-array $A[i] \ldots A[i]$. 

![Insertion Sort Example]

- The **red** item is sorted, and the **green** item is inserted into its correct position.
Insertion Sort

- Go through the items in the array (list) one by one
- For each item $x$ at index $i$:
  - We know the sub-array $A[0] \ldots A[i - 1]$ is sorted
  - **Insert** $x$ in its correct position in the sub-array $A[i] \ldots A[i]$. 

![Insertion Sort Diagram]

[Diagram showing the insertion of elements into a sorted array.]

```plaintext
4 3 2 10 12 1 5 6
```

```plaintext
3 4 2 10 12 1 5 6
```
Insertion Sort

- Go through the items in the array (list) one by one
- For each item $x$ at index $i$:
  - We know the sub-array $A[0] \ldots A[i-1]$ is sorted
  - **Insert** $x$ in its correct position in the sub-array $A[i] \ldots A[i]$. 

![Insertion Sort Diagram]

- **Diagrams** showing the process of sorting the array.
Insertion Sort

- Go through the items in the array (list) one by one
- For each item \( x \) at index \( i \):
  - We know the sub-array \( A[0] \ldots A[i-1] \) is sorted
  - **Insert** \( x \) in its correct position in the sub-array \( A[i] \ldots A[i] \).
Insertion Sort

- Go through the items in the array (list) one by one
- For each item $x$ at index $i$:
  - We know the sub-array $A[0] \ldots A[i-1]$ is sorted
  - **Insert** $x$ in its correct position in the sub-array $A[i] \ldots A[i]$. 

![Diagram showing the insertion sort process](image)
Insertion Sort

- Go through the items in the array (list) one by one
- For each item $x$ at index $i$:
  - We know the sub-array $A[0] \ldots A[i-1]$ is sorted
  - Insert $x$ in its correct position in the sub-array $A[i] \ldots A[i]$. 

---

![Diagram of Insertion Sort Process](image-url)
Insertion Sort

- Go through the items in the array (list) one by one
- For each item $x$ at index $i$:
  - We know the sub-array $A[0] \ldots A[i-1]$ is sorted
  - **Insert** $x$ in its correct position in the sub-array $A[i] \ldots A[i]$. 
Insertion Sort

- Go through the items in the array (list) one by one
- For each item $x$ at index $i$:
  - We know the sub-array $A[0] \ldots A[i-1]$ is sorted
  - Insert $x$ in its correct position in the sub-array $A[i] \ldots A[i]$. 

![Diagram of Insertion Sort process]
# Insertion Sort

/* Function to sort an array using insertion sort */

void insertionSort(int arr[], int n)
{
    int i, key, j;
    for (i = 1; i < n; i++)
    {
        key = arr[i];
        j = i - 1;

        /* Move elements of arr[0..i-1], that are greater than key, to one position ahead of their current position */
        while (j >= 0 && arr[j] > key)
        {
            arr[j+1] = arr[j];
            j = j - 1;
        }
        arr[j+1] = key;
    }
}
Insertion Sort Summary

- **One Iteration of the Insertion Sort Algorithm:**
  - After the $i$th iteration, $A[0..i]$ is sorted.
  - Insert item $A[i+1]$ in its proper place in $A[0..i]$.
Insertion Sort Summary

- **One Iteration of the Insertion Sort Algorithm:**
  - After the $i$th iteration, $A[0..i]$ is sorted.
  - Insert item $A[i + 1]$ in its proper place in $A[0..i]$.

- In the worst case, $i$ items are moved in the $i + 1$th iteration!
In the worst case the array is sorted backwards.

\[ \begin{array}{cccccc}
  n & n-1 & n-2 & \ldots & 3 & 2 & 1 \\
  \vdots & & & & 3 & 2 & 1 \\
  3 & 4 & 5 & \ldots & n & 2 & 1 \\
  2 & 3 & 4 & \ldots & n-1 & n & 1 \\
  1 & 2 & 3 & \ldots & n-2 & n-1 & n \\
\end{array} \]
Insertion Sort Analysis

- In the worst case the array is sorted backwards.

```
| n  | n-1  | n-2  | ... | 3  | 2  | 1 |
```

```
|     |     |     | ... | 3  | 2  | 1 |
```

```
|     |     |     | ... | 3  | 2  | 1 |
```

```
|     |     |     |     |    |    | 1 |
```

```
| 3  | 4  | 5  | ... | n  | 2  | 1 |
```

```
| 2  | 3  | 4  | ... | n-1| n  | 1 |
```

```
| 1  | 2  | 3  | ... | n-2| n-1| n |
```

- The total number of moved items:

\[ 1 + 2 + \ldots + n - 1 = \frac{n(n-1)}{2} \in (n^2) \]
The worst-case running time of insertion sort is $O(n^2)$. 
### Insertion Sort Time Complexity

- The **worst-case** running time of insertion sort is $O(n^2)$.
- As it turns out, the **average-case** running time is also $O(n^2)$.
- Faster sorting algorithms exist. These include:

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>worst case</th>
<th>average case</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quicksort</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Merge Sort</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Heapsort</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The **worst-case** running time of insertion sort is $O(n^2)$.

As it turns out, the **average-case** running time is also $O(n^2)$.

Faster sorting algorithms exist. These include:

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>worst case</th>
<th>average case</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quicksort</td>
<td>$O(n^2)$ (random pivot)</td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td>Merge Sort</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td>Heapsort</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
</tr>
</tbody>
</table>
The worst-case running time of insertion sort is $O(n^2)$. As it turns out, the average-case running time is also $O(n^2)$.

Faster sorting algorithms exist. These include:

<table>
<thead>
<tr>
<th></th>
<th>worst case</th>
<th>average case</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quicksort</td>
<td>$O(n^2)$ (random pivot)</td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td>Merge Sort</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td>Heapsort</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
</tr>
</tbody>
</table>

The lower bound on the worst-case time complexity of any comparison-based sorting algorithm is also $n \log n$. 