CMPT 225

Lecture 11 – Simple sorting algorithms
Last Lecture

- We saw how to ...
  - Describe Queue
  - Define public interface of Queue ADT
  - Design and implement Queue ADT using various data structures
  - Compare and contrast these various implementations using Big O notation
  - Give examples of real-life applications (problems) where we could use Queue to solve the problem
  - Solve problems using Queue ADT
Learning Outcomes

- At the end of the next few lectures, a student will be able to:
  - describe the behaviour of and implement simple sorting algorithms:
    - insertion sort
    - selection sort
  - describe the behaviour of and implement more efficient sorting algorithms:
    - quick sort
    - merge sort
  - analyze the best, worst, and average case running time (and space) of these sorting algorithms
Today’s menu

Looking at

- insertion sort
- selection sort

Analyze their best, worst, and average case running time and space efficiency of these sorting algorithms
Why Sorting?

- **Definition**: Process of placing elements in a particular sort order based on the value of a/some search key(s)
  - Ascending/descending sort order
- **Why sorting?**
  - Easier to deal with sorted data: easier to search (e.g. binary search)
  - Common operation but time consuming
- **What can be sorted?**
  - Internal data (data fits in memory)
  - External data (data that must reside on secondary storage)
- **How to sort?**
Selection Sort
How Selection Sort works

- Array has \( n \) elements
- Starts with element at index 0 and ends with element at index \( n - 1 \)

- Until the array is sorted
  1. Find (select) the smallest element in the unsorted section of array
     - This is done by comparing 1 element with all \( n-1 \) other elements
  2. Swap it with the first element in the unsorted section of array
Let’s have a look at Selection Sort
Selection Sort is **in place** algorithm

- **in-place**: algorithm does not require additional array(s)
- Selection sort starts with an unsorted array:

![Unsorted Array]

- As the array is being sorted, the unsorted section decreases and sorted section increases:

  ![Sorting Process]

- ...
Time Efficiency Analysis of Selection Sort - 1

<table>
<thead>
<tr>
<th>Unssorted elements</th>
<th>Number of comparisons required to select “the one” (smallest or largest)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( n )</td>
<td>( n-1 )</td>
</tr>
<tr>
<td>( n-1 )</td>
<td>( n-2 )</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>( n(n-1)/2 )</td>
<td></td>
</tr>
</tbody>
</table>
In total, selection sort ...
- Makes $n(n-1)/2$ comparisons
- Performs $n-1$ swaps

Would the way the data is organized affect the number of operations selection sort perform (affect its time efficiency)?
- For example:
  - If the data was already sorted (in the desired sort order, e.g., ascending)?
  - If the data was sorted but in the other sort order (e.g., descending)?
  - If the data was unsorted?
Summary – Selection Algorithm

- Time efficiency
  - Best case scenario:
  - Average case scenario:
  - Worst case scenario:

- Space efficiency
  - Best case scenario:
  - Average case scenario:
  - Worst case scenario:
Insertion Sort
How Insertion Sort works

- Array has $n$ elements
- At the start, insertion sort considers the first cell of array to be already sorted -> sorted section
- So, it actually starts with element at index 1 and ends with last element (at index $n-1$)
- Until the array is sorted

1. Pick the 1$^{st}$ element of the unsorted section and insert it its correct (sorted) place in the sorted section of array
   - This is done by comparing the 1$^{st}$ element of the unsorted section with each element of the sorted section
2. Shift the elements in sorted section up one position to make space for 1$^{st}$ element of unsorted section of array (if needed)
3. Inserts the element in correct position in sorted section
Let’s have a look at Insertion Sort
Insertion Sort is **in place** algorithm

- **in-place**: algorithm does not require additional array(s)
- Insertion sort starts with an unsorted array:

  ![Unsorted Array Diagram]

- As the array is being sorted, the unsorted section decreases and sorted section increases:

  ![Sorting Process Diagram]

  ![Partial Sorted Array Diagram]
## Time Efficiency Analysis of Insertion Sort - 1

<table>
<thead>
<tr>
<th>Sorted Elements</th>
<th>Worst-case Comparison</th>
<th>Worst-case Shift</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>(n-1)</td>
<td>(n-1)</td>
<td>(n-1)</td>
</tr>
<tr>
<td>(n(n-1)/2)</td>
<td>(n(n-1)/2)</td>
<td></td>
</tr>
</tbody>
</table>
Time efficiency of insertion sort is affected by the way data is organized in the array to be sorted.

In the best case scenario, the array is

- Requires \( n - 1 \) comparisons
- No shift is required
Time Efficiency Analysis of Insertion Sort - 3

- In the **worst case scenario**, the array is
  - Every element has to be moved
  - Every element in sorted section of array has to be shifted
  - The outer loop runs $n-1$ times
    - In the first iteration, one comparison and shift
    - In the last iteration, $n-1$ comparisons and shifts
    - On average, $(n \ (n-1) / 2) / (n-1) = n/2$ comparisons and shifts
  - For a total of $(n-1) \ast n/2$ comparisons and shifts
What is the **average case scenario**?

- If array contains totally unsorted data, insertion sort is usually closer to the worst case scenario
Summary – Insertion Algorithm

- Time efficiency
  - Best case scenario:
  - Average case scenario:
  - Worst case scenario:

- Space efficiency
  - Best case scenario:
  - Average case scenario:
  - Worst case scenario:
Summary – Simple Sorting Algorithms

- Insertion sort
  - **Efficient**: for small \( n \)'s
    - More efficient in practice than most other simple quadratic (i.e., \( O(n^2) \)) algorithms
  - **Stable**: does not change the relative order of elements with equal keys

- Both sorts
  - **In-place**: only requires a constant amount \( O(1) \) of additional memory space
Learning Check

- We can now ...
  - Define the best, worst, and average case running time and space efficiency of
    - insertion sort
    - selection sort
  - Considering the “simple” sorting algorithms, identify the most efficient one and explain why it is the most efficient
Next Lectures

- More efficient sorting algorithms