Algorithms: Sorting

Instructor: Nihshanka Debroy
Running time

• Worst-case running time
  – upper bound on the running time
  – guarantee the algorithm will never take longer to run

• Average-case running time
  – time it takes the algorithm to run on average (expected value)

• Best-case running time
  – lower bound on the running time
  – guarantee the algorithm will not run faster
Sorting

- Given $n$ items, rearrange them so that they are in increasing order
- A key recurring problem
- Many different methods, how do we choose?
- Given a set of cards, describe how you would sort them:

- Given a set of words, describe how you would sort them in alphabetical order?
Selection Sort

- Starts by looking for **smallest element** in array
- If the smallest entry is in position i in the array, algorithm **swaps** value in position i with the value in position 0.
- Done by using an extra variable to help with the swap:
  ```
  int temp = array[0];
  array[0] = array[i];
  array[i] = temp;
  ```
- After this first swap, find second smallest entry and swap into position 1. Then find third smallest and swap, then the fourth smallest and swap, and so on.
Selection Sort: Correctness and Running Time

- **Step i**: value that should be in position i of sorted array is found and swapped into position i

- **Swap** is done with some array position with index greater than i (as array positions \( \text{array}[0 \ldots i-1] \) are already in their correct positions)

- So, after looping through \( n \) iterations, for an array with \( n \) elements, we have a sorted array.

- **Running time**: \( O(n + (n-1) + (n-2) + \ldots 1) = O(n^2) \)

- Do same thing regardless of initial array.
Insertion Sort

Example from cards:

➢ Start with a section of card we have sorted.
➢ Then add or "insert" one more card into its proper place in that sorted section.
➢ Now we have a slightly larger sorted section.
➢ Eventually, after all cards are inserted into their place one after another entire hand of cards is sorted.
Insertion Sort: Algorithm

- Start each iteration with a subsection of the array already sorted. (When the algorithm first starts, the first element by itself is our "sorted" subsection.)
- Then take first array entry outside this sorted subsection and swap it down the line to correct position => continues until a point where element before it is less than or equal to it and element after it is larger.
- With this element inserted, sorted subsection's size increased by 1.
- After \( n \) such iterations for an array with \( n \) elements, entire array is sorted.
Insertion Sort: Running Time

- Only swap next element that we want to insert as many times as we have to.
- If it is smaller than all elements already in sorted subsection - lot of swaps!
- If it is larger than all of those elements we do not have to make any swaps.
- Case where element being inserted is new smallest element causes worst case behavior - have to make a swap for every element already in our sorted subsection.
- **Worst case running time:** $O(1 + \ldots + (n-2) + (n-1)) = O(n^2)$
- Case where element to be inserted is larger than all elements in the sorted subsection – no swaps needed. If each iteration like this: best case behavior, running time of only $O(n)$ – happens if array was already sorted when given to us.
Comparisons in insertion sort

- **Worst case**
  - element $k$ requires $(k-1)$ comparisons
  - total number of comparisons:
    \[0 + 1 + 2 + \ldots + (n-1) = \frac{1}{2} (n)(n-1) = \frac{1}{2} (n^2-n)\]

- **Best case**
  - elements 2 through $n$ each require one comparison
  - total number of comparisons:
    \[1 + 1 + 1 + \ldots + 1 = n-1\]
Running time of insertion sort

- Best case running time is *linear*
- Worst case running time is *quadratic*
- Average case running time is also *quadratic*
  - on average element k requires \((k-1)/2\) comparisons
  - total number of comparisons:
    \[
    \frac{1}{2} \left( 0 + 1 + 2 + \ldots + n-1 \right) = \frac{1}{4} (n)(n-1)
    = \frac{1}{4} (n^2-n)
    \]
Merge Sort
## Merging two sorted lists

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>first list</strong></td>
<td><strong>second list</strong></td>
<td><strong>result of merge</strong></td>
</tr>
<tr>
<td>10 27</td>
<td>12 20</td>
<td>10</td>
</tr>
<tr>
<td>10 27</td>
<td>12 20</td>
<td>10 12</td>
</tr>
<tr>
<td>10 27</td>
<td>12 20</td>
<td>10 12 20</td>
</tr>
<tr>
<td>10 27</td>
<td>12 20</td>
<td>10 12 20 27</td>
</tr>
</tbody>
</table>
Mergesort

```
27 10 12 20
     divide
  27 10        12 20
     divide     divide
  27 10        12 20
     merge    merge
  10 27        12 20
     merge
  10 27 20
  10 12 20 27
```
Merge Sort

- Uses **Divide-and-Conquer** approach
- Divide array into halves at each step
- **$\lg n$ steps** to get down to single elements
- At most $n$ comparisons at each merge
- **$\lg n$ merges**
- Running time: $O(n \lg n)$