CS21: INTRODUCTION TO COMPUTER SCIENCE

Prof. Mathieson
Fall 2018
Swarthmore College
Outline Nov 16:

• Quiz 4: first 25-30min
• Runtime of bubble sort and selection sort
• Insertion Sort demo
• Runtimes in action
• Can we create a faster sorting algorithm?

Notes

• Lab 8 due tomorrow night!
• Office hours today 3-5pm, Ninja session tonight
• Lab 9 posted soon, due Mon after Thanksgiving
3 sorts for this class

- **Selection Sort**: iteratively find the minimum element and place it in the correct position

- **Bubble Sort**: move through the list, swapping adjacent elements if they are out of place (repeat until sorted)

- **Insertion Sort (didn’t get to)**: for each element of the list, move it down until it is in the correct position
Types of sorting

- **Out-of-place**: leaves the original list alone and creates a new sorted list (returns new list)

- **In-place**: modifies (mutates) the original list via swaps so that it is now sorted

- **Pros of in-place sort**: no new data structure needed (saving space)

- **Cons of in-place sort**: original order destroyed (in some cases it might be important), can be more difficult to implement
Runtime of our sorting algorithms

Bubble Sort

\[ \sum_{j=0}^{n-1} j = \frac{n(n-1)}{2} \]

\[ \sum \text{of comparisons} = \frac{n^2 - n}{2} \]

\[ O(n^2) \]

\[ \text{Order} \]

\[ \text{Sum} \]

\[ O(n) \]

\[ \text{Runtime} \]
Insertion Sort demo

- [https://visualgo.net/bn/sorting](https://visualgo.net/bn/sorting)
Runtime Comparison demo

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

<table>
<thead>
<tr>
<th></th>
<th>Insertion</th>
<th>Selection</th>
<th>Bubble</th>
<th>Shell</th>
<th>Merge</th>
<th>Heap</th>
<th>Quick</th>
<th>Quick3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Play All</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Random</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Nearly Sorted</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reversed</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Few Unique</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Runtime in action

- /cs21/inclass/w10/sort_runtime.py

- Idea: if we double the length of the list, we should see the runtime quadruple (x4)
sort_runtime.py example output

<table>
<thead>
<tr>
<th>Number of elements</th>
<th>Runtime of selection sort (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000</td>
<td>0.15</td>
</tr>
<tr>
<td>4000</td>
<td>0.55</td>
</tr>
<tr>
<td>8000</td>
<td>2.16</td>
</tr>
<tr>
<td>16000</td>
<td>8.81</td>
</tr>
<tr>
<td>32000</td>
<td>35.08</td>
</tr>
</tbody>
</table>
Can we do better than $O(n^2)$?

- Idea: thinking along the lines of binary search, what if we could divide the list in half and sort both pieces, then merge them together?