Outline for this week

• HW5 due today via Box (3 of you already did)
• Quiz 6 tomorrow in recitation, will cover up to Thursday's lecture (Interfaces, Iterators, Stacks/Qs)
• Next HW out Thursday, due April 9
• Searching & Sorting, Tree intro
Searching (linear/sequential search)

• How did we implement contains?
  – start at first element, go until you find it (or not)

• Analysis?
  • Worst case $O(n)$ - item in last position/not found
  • So, if my search space doubles in size, I need to look at twice as many elements in the worst case

• Can we do better?
  – yes, but…
Searching (binary search)

• Start at middle element, determine whether you're smaller/bigger and search the left/right half

• Analysis?
  • Worst case - each stage cuts the search space in half. How many times can we halve a space of size n? \( O(\log n) \)
  • So, if my search space doubles in size, I need only look at one additional element!

• But (there's always a but)…
  – the collection needs to be in order (and there’s a cost)!
  – AND must have constant-time access to each element
Timing (Searches)

• Why do we use O notation (Big O)?
  – Because we need a machine/OS-independent way of measuring algorithm performance
  – **BUT**, if you’re running on your machine, you can do absolute timings, no?
  – Of course, different machines will have different timings, but on your machine, it should be consistent

• So let's time some searches…
Sorting

• In order to sort, you need to be able to compare elements (a total order):
  – total:  $a R b$ or $b R a$,
  – transitive:  $a R b$ and $b R c \implies a R c$
  – anti-symmetric:  $a R b$ and $b R a \implies a == b$

• Types
  – ints, doubles:  easy
  – Objects?  must be Comparable

• So, how do you sort?
Sorting (Bubble sort)

• Bubble Sort = = BS
• Algorithm/Illustrate
  – Start at the first element and pair-wise compare adjacent elements; if they are out of order, swap them
  – Repeat this until you go through all the elements and don't make a swap
  – *Invariant:* After the $i^{th}$ pass, the $i^{th}$ largest elements are in their final position (a "sinking" sort)
• Analysis?
  – Worst case, does $n-1$ compares (AND swaps), then $n-2$, then $n-3$…
  – $O(n^2)$ comparisons AND $O(n^2)$ swaps --> not good!
Sorting (Insertion sort)

- Algorithm/Illustrate (like sorting a hand of cards)
  - In each iteration, i, take the element in the i\textsuperscript{th} position, compare it to the one before until you find the place where it belongs (while it's less than the one ahead of it, move the one ahead down, making a hole; not really “swapping”)
  - \textit{Invariant}: After the i\textsuperscript{th} pass, the first i elements are sorted relative to each other

- Analysis?
  - O(n) to go through all the elements
  - O(n) worst case (which is?) to find the place to insert element
  - O(n) * O(n) --> O(n\textsuperscript{2}) comp’s, but in practice often pretty fast
Sorting (Selection sort)

- Algorithm/Illustrate
  - In each iteration, i, select the i\textsuperscript{th} smallest element and store it in the i\textsuperscript{th} position
  - \textit{Invariant}: After the i\textsuperscript{th} pass, the i\textsuperscript{th} smallest elements are in their final position

- Analysis?
  - O(n) to find the smallest element in n objects
  - Must go through n-1 positions of the collection: O(n)
  - O(n) * O(n) --> O(n^2) comparisons

- Can we do better than O(n^2)?
Sub-Quadratic Sorts

• Mergesort
• Quicksort
• Radix sort
Sorting (Merge sort)

- Algorithm/Illustrate (w/numbers)
  - Divide the collection in half
  - Recursively sort the two halves (by calling mergesort)
  - Merge the halves back together
  - *Invariant*: the merged “halves” are sorted
Sorting (Merge sort)

• Analysis?
  – $O(\log n)$ - the number of times you can divide in half
  – $O(n)$ - the time to merge two halves into a whole
  – $O(\log n) \times O(n) \rightarrow O(n \log n)$
  – BUT (see previous note about "but"), this algorithm needs a separate, auxiliary, array to store the halves
Sorting (Quicksort)

- **Algorithm/Illustrate (w/numbers)**
  - "Randomly" pick an element about which to partition the collection into two parts
  - Partition the array around that value, called the pivot, so that the partition value ends up in final position, i.e., the array looks like: < pivot, pivot, >= pivot
  - Recursively sort the two parts (by calling quicksort)
  - **Invariant**: After the $i^{th}$ pass, the $i^{th}$ partition value/pivot is in its final position (i.e., all values to the left are less than the partition value/pivot and all the values to the right are greater than or equal to the partition value/pivot)
Sortng (Quicksort)

- Analysis? Well…
  - $O(n \log n)$ if all goes well in choosing the pivot - which is when what is true about where the pivot ends up?
  - BUT, $O(n^2)$ in worst case - when might that be?
  - In practice, though, usually $O(n \log n)$ and faster (better constant) than merge sort (and no need for auxiliary array)
Sorting (can we do better than $n \log n$?)

- **Bucket sort**
  - Algorithm - need a bucket for each possible value
  - Analysis
    - if $O(1)$ to insert an element into a bucket --> $O(n)$ to insert all elements
    - if $O(m)$ to collect all the buckets --> $O(n + m)$ overall
  - Limitation: finite number of possible values (finite buckets)

- **Radix sort for integers**
  - 10 buckets (0 - 9); sort integers into appropriate bucket starting with least significant digit (int % 10), collect them and sort by next least significant digit until out of digits
  - $O(n \times k \times 10)$ where $k$ is number of digits --> $O(n)$ overall
Sorting (Stable sorts)

- Definition
  - A stable sort maintains the relative position of equal elements
  - Benefit? If you were sorting students and sorted by name and then by gender, then you’d get a list that was sorted by gender, but alphabetical within gender

- Which sorts preserve stability?
  - Insertion sort
  - Merge sort

- And the others
  - Selection sort - no (why?)
  - Quicksort – no, for similar reason as selection sort
## Sorting (summary)

<table>
<thead>
<tr>
<th>Sort</th>
<th>Best case</th>
<th>Average</th>
<th>Worst case</th>
<th>Stable?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selection</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Insertion</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Merge</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Quicksort</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
# Sorting (summary)

<table>
<thead>
<tr>
<th>Sort</th>
<th>Best case</th>
<th>Average</th>
<th>Worst case</th>
<th>Stable?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selection</td>
<td>O(n²)</td>
<td>O(n²)</td>
<td>O(n²)</td>
<td>no</td>
</tr>
<tr>
<td>Insertion</td>
<td>O(n)</td>
<td>O(n²)</td>
<td>O(n²)</td>
<td>yes</td>
</tr>
<tr>
<td>Merge</td>
<td>O(n log n)</td>
<td>O(n log n)</td>
<td>O(n log n)</td>
<td>yes</td>
</tr>
<tr>
<td>Quicksort</td>
<td>O(n log n)</td>
<td>O(n log n)</td>
<td>O(n²)</td>
<td>no</td>
</tr>
</tbody>
</table>