Outline

- Quadratic-Time Sorting
  - Bubble/Shaker Sort
  - Insertion Sort
  - Odd-Even Sort

- Linearithmic-Time Sorting
  - Heap Sort
  - Merge Sort
  - Quick Sort

- Conclusion

Check out this link for animation of various sorting algorithms:
http://cs.pugetsound.edu/~aasmith/sorters/
Heap Sort

- Already saw this: (Refer back to Lec 8 notes)
  - Insert all elements in the list into a min-heap
  - Repeat until heap is empty
    - Remove smallest element from the heap
    - Put it in tail of new list

```java
public static int[] heapSort(int[] list) {
    Heap<Integer> heap = new Heap<>();

    // build min-heap -- add all items from the list
    for (int i = 0; i < list.length; i++) {
        heap.add(list[i]);
    }

    // re-build the list by continuously pulling the min from the heap
    for (int i = 0; i < list.length; i++) {
        list[i] = heap.remove(0);
    }

    return list;
}
```
Heap Sort Analysis

- **Pros:**
  - It's fast: $O(n \log n)$ time in best/worst/average case

- **Cons:**
  - *Our* version uses up $n$ space (need to create a heap of size $n$)
  - But there is a better version that does not create auxiliary space
Outline

- Quadratic-Time Sorting
  - Bubble/Shaker Sort
  - Insertion Sort
  - Odd-Even Sort

- Linearithmic-Time Sorting
  - Heap Sort
  - Merge Sort
  - Quick Sort

- Conclusion

Check out this link for animation of various sorting algorithms:
http://cs.pugetsound.edu/~aasmith/sorters/
To merge two sorted arrays into a new sorted array:

- Repeatedly compare the two smallest items
- Copy the smaller of the two to the new array
Define merge() First

```java
public static int[] merge(int[] first, int[] second) {
    // merge the two pre-sorted lists
    int[] merged = new int[first.length + second.length];
    int i = 0, j = 0, k = 0;
    while (j < first.length && k < second.length) {
        if (first[j] <= second[k]) {
            merged[i++] = first[j++];
        } else {
            merged[i++] = second[k++];
        }
    }
    // finish off any stragglers from first list
    while (j < first.length) {
        merged[i++] = first[j++];
    }
    // finish off any stragglers from second list
    while (k < second.length) {
        merged[i++] = second[k++];
    }
    return merged;
}
```
Merging Two Sorted Arrays

- To merge two sorted arrays into a new sorted array:
  - Repeatedly compare the two smallest items (tiebreak: take from first)
  - Copy the smaller of the two to the new array

- Complexity: $O(m + n)$, where $m, n$ refer to sizes of the 2 arrays
Merge Sort

- Observations:
  - Just established merging 2 sorted lists into a sorted list takes linear time
  - A list containing one element is sorted

- *Merge Sort* exploits the above two observations
  - Input an unsorted list
  - Recursively split the list in two unsorted halves
    - Until each of the halves contains one element
    - Merge the sorted halves together
/**
 * Merge Sort
 * @param list a list of ints
 * @return a sorted list of ints
 */
public static int[] mergeSort(int[] list) {
    // base case: list is sorted when empty or contains one item
    if (list.length <= 1) {
        return list;
    }

    int midpt = list.length/2;

    // first half of list
    int[] first = Arrays.copyOfRange(list, 0, midpt);
    first = mergeSort(first);  // sort it

    // second half of list
    int[] second = Arrays.copyOfRange(list, midpt, list.length);
    second = mergeSort(second);  // sort it

    // merge the sorted lists
    return merge(first, second);
}
Merge Sort (Cont.)

Recursive call to mergeSort()

Return from previous call to mergeSort()
Merge Sort (Cont.)

Recursive call to mergeSort()

Return from previous call to mergeSort()
Merge Sort (Cont.)

Recursive call to mergeSort()

Return from previous call to mergeSort()
Recursive call to mergeSort()

Return from previous call to mergeSort()
Merge Sort (Cont.)

Recursive call to mergeSort()

Return from previous call to mergeSort()
Merge Sort (Cont.)

Recursive call to mergeSort()

Return from previous call to mergeSort()
Merge Sort (Cont.)

Recursive call to mergeSort()

Return from previous call to mergeSort()
Merge Sort (Cont.)

```
8 5 1 3 0 5 8 0
8 5 1 3
5 8
1 3
8 5 1
---
```

- Recursive call to mergeSort()
- Return from previous call to mergeSort()
Merge Sort (Cont.)

Recursive call to mergeSort()

Return from previous call to mergeSort()
Merge Sort (Cont.)

Recursive call to mergeSort()

Return from previous call to mergeSort()
Merge Sort (Cont.)

```
8 5 1 3 0 5 8 0
```

```
merge() 1 3 5 8
merge() 5 8
merge() 1 3
8 5 1 3
```

- Recursive call to mergeSort()
- Return from previous call to mergeSort()
Merge Sort (Cont.)

Recursive call to mergeSort()

Return from previous call to mergeSort()
Merge Sort (Cont.)

Recursive call to mergeSort()

Return from previous call to mergeSort()
Merge Sort (Cont.)

Recursive call to mergeSort()

Return from previous call to mergeSort()
Merge Sort (Cont.)

Recursive call to mergeSort()

Return from previous call to mergeSort()
Merge Sort (Cont.)

Recursive call to mergeSort()

Return from previous call to mergeSort()
Merge Sort (Cont.)

![Diagram of Merge Sort process]

- Merge Sort is a recursive algorithm that divides the list into two halves, sorts them, and then merges them back together.
- The diagram shows the recursive call to `mergeSort()` and the return from the previous call to `mergeSort()`.

Recursive call to `mergeSort()`
Return from previous call to `mergeSort()`
Merge Sort (Cont.)

Recursive call to mergeSort()

Return from previous call to mergeSort()
Merge Sort (Cont.)

Recursive call to mergeSort()

Return from previous call to mergeSort()
Merge Sort (Cont.)

Recursive call to mergeSort()

Return from previous call to mergeSort()
Merge Sort (Cont.)

```
8 5 1 3 0 5 8 0

merge() 1 3 5 8
merge() 0 0 5 8
merge() 5 8
merge() 1 3
merge() 0 5
merge() 0 8
merge() 8
merge() 5
merge() 1
merge() 3
merge() 0
```
Merge Sort (Cont.)

Recursive call to mergeSort()

Return from previous call to mergeSort()
Class Activity: MergeSort

- Given: A list of numbers someone asked me to sort
  - If list is empty or has one item,
    - Hand it back to the student who gave you the list
  - Else:
    - Divide list in two halves (if it doesn't divide evenly, give extra item to the left list)
    - Turn around (and you should be facing 2 students behind you)
      - Hand off the left list to a student
        » Wait for them to hand it back to you sorted
      - Hand off the right list to a student
        » Wait for them to hand it back to you sorted
    - Once you receive both halves, merge() them, and hand it forward!
Complexity of Merge

- First, determine the time complexity of `merge()`
  - Just need to examine each item once
  - As we know from before, \[ T_{merge}(m, n) = m + n \]

- But we're always merging two roughly-equal halves:
  - So, in our case, \[ T_{merge}(n/2, n/2) = n \]
Complexity of MergeSort

- Intuitively....

Problem Size:

- $n$
- $n/2$
- $n/4$
- ...
- 1

Steps, $T(n)$, per level

- $2n+1$ (why?)
- $2[2(n/2)+1]+1 = 2n+2$
- $4[2(n/4)+1]+2+1 = 2n+4$
- ...
- 1

MergeSort's time complexity:

$$T(n) = \begin{cases} 1, & \text{if } n \leq 1 \\ 2T(n/2) + 2n + 1, & \text{otherwise} \end{cases}$$
Complexity of MergeSort

- Intuitively....

<table>
<thead>
<tr>
<th>Problem Size</th>
<th>Steps, T(n), per level</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>2n+1 (why?)</td>
</tr>
<tr>
<td>n/2</td>
<td>2[2(n/2)+1]+1 = 2n+2</td>
</tr>
<tr>
<td>n/4</td>
<td>4[2(n/4)+1]+2+1 = 2n+4</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

- MergeSort's time complexity:

\[
T(n) = \begin{cases} 
1, & \text{if } n \leq 1 \\
2T(n/2) + 2n + 1, & \text{otherwise}
\end{cases}
\]

\[
T(n) = O(n \log_2 n)
\]
Analysis of MergeSort

- **Pros:**
  - Linearithmic time complexity
  - Time complexity is consistent (best/avg/worst) and predictable

- **Cons:**
  - Space complexity can be a problem
    - Requires 2 additional copies of the list each recursive call
      - One created when splitting, another created when merging
  - Semi-sorted lists don't help, unlike the linear-time sorts
Outline

- Quadratic-Time Sorting
  - Bubble/Shaker Sort
  - Insertion Sort
  - Odd-Even Sort

- Linearithmic-Time Sorting
  - Heap Sort
  - Merge Sort
  - Quick Sort

- Conclusion

Check out this link for animation of various sorting algorithms:
http://cs.pugetsound.edu/~aasmith/sorters/
Quick Sort

- Input: two indices, start and end, of current list
  - If the list isn't empty (\(i.e., \) start < end):
    - Use last item in the list as the pivot
      - Rearrange (partition) all items in the list so that:
        - Items greater than it appear on its right
        - Items less than pivot appear on its left
    - Then recurse on both unsorted sublists
Pivot and Partitioning

- Trace for: 7 3 9 5 1 3 5 and for 6 4 2 1 0

```java
/**
 * Rearranges items in the list and returns the pivot's position
 */
private static int partition(int[] list, int start, int end) {
    int pivot = list[end]; // use last item in the sublist as pivot
    int i = start;
    for (int j = start; j < end; j++) {
        if (list[j] < pivot) {
            swap(list, i, j);
            i++;
        }
    }
    // swap pivot into proper position
    swap(list, i, end);
    return i;
}

private static void swap(int[] list, int i, int j) {
    int tmp = list[i];
    list[i] = list[j];
    list[j] = tmp;
}
```
Complexity of Partition

- Determine the time complexity of partition()
  - Select a pivot item
  - Examine each item, comparing with pivot
    - Swapping, if needed
  - One last swap to get pivot in place
  - Return list

- Complexity: $T_{part}(n) \approx 4n + 4$
QuickSort

```java
/**
 * QuickSort
 * @param list  A list of ints
 * @return a sorted list of ints
 */
public static int[] quickSort(int[] list) {
    return quickSort(list, 0, list.length-1);
}

private static int[] quickSort(int[] list, int start, int end) {
    if (start < end) {
        // partitions the list, and gets the position of the pivot
        int pivot_idx = partition(list, start, end);

        // sort the two other halves
        quickSort(list, start, pivot_idx-1);
        quickSort(list, pivot_idx+1, end);
    }
    return list;
}
```
Quick Sort Trace

List

8 6 9 5 1 3 2 4
Quick Sort Trace: First partition(list, 0, 7)

```python
quicksort(list, 0, 7)
```

List

```
8 6 9 5 1 3 2 4
```

pivot
Quick Sort Trace: First partition(list, 0, 7)

List

8 6 9 5 1 3 2 4

pivot

quicksort(list, 0, 7)

8 6 9 5 1 3 2 4
Quick Sort Trace: First partition(list, 0, 7)

```
List: 1 6 9 5 8 3 2 4
```

**swap**

**pivot**

```
quicksort(list, 0, 7)
```

```
1 6 9 5 8 3 2 4
```
Quick Sort Trace: First partition(list,0,7)
Quick Sort Trace: First partition(list,0,7)

List 1 6 9 5 8 3 2 4 4

pivot swap

quicksort(list, 0, 7)

1 6 9 5 8 3 2 4
Quick Sort Trace: First partition(list,0,7)

```
quicksort(list, 0, 7)
```

```plaintext
1 3 9 5 8 6 2 4
```

List

```
1 3 9 5 8 6 2 4
```

i

j

pivot

swap
Quick Sort Trace: First partition(list,0,7)

Quick Sort Trace: First partition(list,0,7)

List:
1 3 9 5 8 6 2 4

i
pivot
j
swap

quicksort(list, 0, 7)

1 3 9 5 8 6 2 4
Quick Sort Trace: First partition(list,0,7)

List

\[\begin{array}{ccccccccc}
1 & 3 & 9 & 5 & 8 & 6 & 2 & 4 \\
\end{array}\]

\text{quicksort}(\text{list}, 0, 7)

\[\begin{array}{ccccccccc}
1 & 3 & 9 & 5 & 8 & 6 & 2 & 4 \\
\end{array}\]
Quick Sort Trace: First partition(list,0,7)

```
Quick Sort

List: 1 3 2 5 8 6 9 4

quicksort(list, 0, 7)
```

```
1 3 2 5 8 6 9 4
```
Quick Sort Trace: First partition(list,0,7)

```
List
1 3 2 5 8 6 9 4
```

**Quick Sort**

```
quicksort(list, 0, 7)
```

```
1 3 2 5 8 6 9 4
```
Quick Sort Trace: First partition(list,0,7)

List: 1 3 2 5 8 6 9 4

Quick Sort Trace: First partition(list,0,7)

Quicksort(list, 0, 7)

Sorted list: 1 3 2 5 8 6 9 4
Quick Sort Trace: First partition(list,0,7)

List: \(1, 3, 2, 5, 8, 6, 9, 4\)

Swap: \(i\) \(j\)

pivot: (Stop!)

\texttt{quicksort}(list, 0, 7)
Quick Sort Trace: First partition(list,0,7)

```
List: 1 3 2 4 8 6 9 5

swap pivot into place!
```

```
quicksort(list, 0, 7)
```

```
List: 1 3 2 4 8 6 9 5
```
Quick Sort Trace: First partition(list, 0, 7)

List

1 3 2 4 8 6 9 5

i

j

swap pivot into place!

quicksort(list, 0, 7)

1 3 2 4 8 6 9 5
Quick Sort Trace: First partition(list, 0, 7)

```plaintext
quicksort(list, 0, 7)
```

List

```
1 3 2 4 8 6 9 5
```

swap pivot into place!
Quick Sort Trace: First partition(list,0,7)

List: 1 3 2 4 8 6 9 5

Swap pivot into place!

quicksort(list, 0, 7)

1 3 2 4 8 6 9 5
Quick Sort Trace (Omitting partition() Trace)

List

quicksort(list, 0, 7)

quicksort(list, 0, 2)

1 2 3 4 8 6 9 5
Quick Sort Trace (Omitting partition() Trace)

List: 1 2 3 4 8 6 9 5

quicksort(list, 0, 7)

quicksort(list, 0, 2)

quicksort(list, 0, 0)
Quick Sort Trace (Omitting partition() Trace)

List 1 2 3 4 8 6 9 5

quicksort(list, 0, 7)

quicksort(list, 0, 2)

quicksort(list, 0, 0)

quicksort(list, 2, 2)
Quick Sort Trace (Omitting partition() Trace)

List: 1 2 3 4 5 6 9 8

quicksort(list, 0, 7)

quicksort(list, 0, 2)

quicksort(list, 4, 7)

quicksort(1 2 3)

quicksort(5 6 9 8)
Quick Sort Trace (Omitting partition() Trace)

List: 1 2 3 4 5 6 8 9

- quicksort(list, 0, 7)
  - quicksort(list, 0, 2)
    - quicksort(list, 0, 1)
      - 1
    - quicksort(list, 2, 2)
      - 2
  - quicksort(list, 3, 7)
    - quicksort(list, 3, 4)
      - quicksort(list, 3, 3)
        - (base case: do nothing)
    - quicksort(list, 5, 7)
      - quicksort(list, 5, 4)
        - quicksort(list, 5, 3)
          - (base case: do nothing)
Quick Sort Trace (Omitting partition() Trace)

List 1 2 3 4 5 6 8 9

quicksort(list, 0, 7)

quicksort(list, 0, 2)

quicksort(list, 4, 7)

quicksort(list, 5, 7)
Quick Sort Trace (Omitting partition() Trace)

List 1 2 3 4 5 6 8 9

quicksort(list, 0, 7)

quicksort(list, 0, 2)  quicksort(list, 4, 7)

quicksort(list, 5, 7)

quicksort(list, 5, 5)
Quick Sort Trace (Omitting partition() Trace)

List: 1 2 3 4 5 6 8 9

quicksort(list, 0, 7)

quicksort(list, 0, 2)

quicksort(list, 4, 7)

quicksort(list, 5, 7)

quicksort(list, 5, 5)

quicksort(list, 7, 7)
Class Activity: QuickSort

- Given: A list of numbers
  - If \( \text{start} \geq \text{end} \)
    - Hand list back to the student who gave it to you
  - Else:
    - Choose \( \text{end} \) element as pivot
    - \textbf{Partition()} the sublist indicated from \text{start} to \text{end}
    - Give left student the list, and:
      - Tell them your \text{start} and \( \text{end} = \text{pivot\_index} - 1 \)
        » Wait to receive the list back
    - Give right student the list, and:
      - Tell them \text{start} = \text{pivot\_index} + 1 and your \text{end}
        » Wait to receive the list back
Another Trace!

List: 5 6 8 9

quicksort(list, 0, 3)

quicksort(list, 4, 3)

base case: do nothing
Another Trace!

```
5 6 8 9
```

```
quicksort(list, 0, 3)
quicksort(list, 0, 2)
quicksort(list, 4, 3)
```

base case: do nothing
Another Trace!

List: [5, 6, 8, 9]

- quicksort(list, 0, 3)
  - quicksort(list, 0, 2)
    - base case: do nothing
  - quicksort(list, 4, 3)
    - base case: do nothing
Another Trace!

List: 5 6 8 9

quicksort(list, 0, 3)

quicksort(list, 0, 2)

quicksort(list, 0, 1)

quicksort(list, 4, 3)

base case: do nothing

quicksort(list, 3, 2)

base case: do nothing
Another Trace!

List

```
5 6 8 9
```

```
quicksort(list, 0, 3)
quicksort(list, 0, 2)
quicksort(list, 0, 1)
quicksort(list, 3, 2)
quicksort(list, 2, 1)
quicksort(list, 4, 3)
```

base case: do nothing
Another Trace!

```
List: 5 6 8 9
```

```
quicksort(list, 0, 3)
quicksort(list, 0, 2)
quicksort(list, 0, 1)
quicksort(list, 0, 0)
quicksort(list, 4, 3)
quicksort(list, 3, 2)
quicksort(list, 2, 1)
```

base case: do nothing
Food for Thought

- Notice how the performance of QuickSort is related to its "Recursion Tree" (results from the trace)

- The first trace, the tree looked more balanced
  - Like MergeSort's recursion tree (which is always balanced)

- The second trace, the tree looked like a linked list
  - Like a worst-case binary search tree...
**Important**: Performance of QuickSort depends on the **pivot**

- `partition()` selects **pivot** as the last item in the sublist

**How to get trees that look like the *first* trace?**

- Want **pivot** to be the median (or near it!) item in the sublist
- Splits the list in half (or close to half!)
Important: Performance of QuickSort depends on the pivot
  • `partition()` selects **pivot** as the last item in the sublist

How to get trees that look like the *first* trace?
  • Want **pivot** to be the median (or near it!) item in the sublist
  • Splits the list in half (or close to half!)
**Important**: Performance of QuickSort depends on the **pivot**
- `partition()` selects **pivot** as the last item in the sublist

How to get trees that look like the **first** trace?
- Want **pivot** to be the median (or near it!) item in the sublist
- Splits the list in half (or close to half!)

\[ n \]

```
partition(list, 0, 7)
```

```
1 3 2 4 8 6 9 5
```

\(~ n/2 \) \(~ n/2 \)
How to get trees that look like the *second* trace?

- When *pivot* is sorted into one of the extreme ends
- Occurs when list is sorted in either ascending or descending order

```plaintext
partition(list, 0, 7)
```
How to get trees that look like the *second* trace?

- When **pivot** is sorted into one of the extreme ends
- Occurs when list is sorted in either ascending or descending order
Pivot: Connection to Performance

- How to get trees that look like the *second* trace?
  - When pivot is sorted into one of the extreme ends
  - Occurs when list is sorted in either ascending or descending order

```
partition(list, 0, 7)
```

```
0 2 2 4 5 5 7 8
```

```
0 2 2 4 5 5
```

```
8
```

```
\sim n - 1
```

```
9 8 5 4 4 3 2 0
```

```
partition(list, 0, 7)
```

```
\sim 1
```
How to get trees that look like the second trace?

- When pivot is sorted into one of the extreme ends
- Occurs when list is sorted in either ascending or descending order

\[ \sim n - 1 \quad \sim 1 \]
How to get trees that look like the *second* trace?

- When *pivot* is sorted into one of the extreme ends
- Occurs when list is sorted in either ascending or descending order
The fix: What if we selected a pivot randomly?

- Instead of always being the last item

RNG says: "Choose 5 as pivot"

Swap pivot to the back

\texttt{partition(list, 0, 7)}
The fix: What if we selected a pivot randomly?

- Instead of always being the last item

RNG says: "Choose 5 as pivot"

Swap pivot to the back

partition(list, 0, 7)
Pivot: Connection to Performance

- **The fix:** What if we selected a **pivot** randomly?
  - Instead of always being the last item

RNG says: "Choose 5 as pivot"

Swap pivot to the back

\[ n \]

\[ \text{partition}(\text{list, 0, 7}) \]

\[ \begin{array}{cccccccc}
0 & 2 & 2 & 4 & 5 & 5 & 8 & 7 \\
\end{array} \]

\[ \begin{array}{cccccccc}
0 & 2 & 2 & 4 & 5 & 5 & 8 & 7 \\
\end{array} \]

\[ \begin{array}{cccc}
0 & 2 & 2 & 4 \\
\end{array} \]

\[ \begin{array}{cccc}
5 & 8 & 7 \\
\end{array} \]

~ \( n/2 \)  ~ \( n/2 \)
The fix: What if we selected a pivot randomly?

- Instead of always being the last item

RNG says: "Choose 5 as pivot"

\[ \text{partition}(\text{list}, 0, 7) \]

\[ \begin{array}{cccccccc}
0 & 2 & 2 & 4 & 5 & 5 & 8 & 7 \\
\end{array} \]

\[ \begin{array}{cccccccc}
0 & 2 & 2 & 4 & 5 & \color{blue}{5} & 8 & 7 \\
\end{array} \]

\[ n \]

\[ \begin{array}{cccc}
0 & 2 & 2 & 4 \\
\end{array} \]

\[ \begin{array}{cccc}
5 & 8 & 7 \\
\end{array} \]

\( \sim n/2 \)

\( \sim n/2 \)

\( \sim n/2 \)

This is awesome!!!
The fix: What if we selected a **pivot** randomly?

- Instead of always being the last item

RNG says: "Choose 8 as pivot"

Swap pivot to the back

```plaintext
partition(list, 0, 7)
```
The fix: What if we selected a pivot randomly?

- Instead of always being the last item

RNG says: "Choose 8 as pivot"

Swap pivot to the back

\[
\text{partition}(\text{list, 0, 7})
\]
The fix: What if we selected a pivot randomly?

- Instead of always being the last item

RNG says: "Choose 8 as pivot"

Swap pivot to the back

```
partition(list, 0, 7)
```

```
0 5 4 4 3 2 8 9
```

```
0 5 4 4 3 2
9
```

\[ \sim n - 1 \]

\[ \sim 1 \]
The fix: What if we selected a pivot randomly?

- Instead of always being the last item

RNG says: "Choose 8 as pivot"

Swap pivot to the back

\[ \text{partition}(\text{list}, 0, 7) \]

\[
\begin{align*}
9 & 8 & 5 & 4 & 4 & 3 & 2 & 0 \\
0 & 5 & 4 & 4 & 3 & 2 & 8 & 9 \\
0 & 5 & 4 & 4 & 3 & 2 & 9 \sim n - 1 \\
& & & & & & & 9 \sim 1
\end{align*}
\]
The fix: What if we selected a pivot randomly?

• Instead of always being the last item

RNG says: "Choose 8 as pivot"

Swap pivot to the back

Or... you can still get pretty unlucky
Analysis of QuickSort

- **Best case:**
  - Median value of the sublist is always picked to be pivot
  - List split in two roughly equal halves

- **Best Case Analysis:**

\[
T(n) = \begin{cases} 
1, & \text{if } n \leq 1 \\
2T(n/2) + T_{part}(n), & \text{otherwise}
\end{cases}
\]

\[
T(n) = O(n \log_2 n)
\]
Analysis of QuickSort

- Worst case:
  - List splits into lopsided "halves"
  - Extreme case is when list size only reduces by 1 item
  - Can be mitigated by using random pivot points

- Complexity:

\[
T(n) = \begin{cases} 
1, & \text{if } n \leq 1 \\
T(n - 1) + T_{\text{part}}(n) + 1, & \text{otherwise}
\end{cases}
\]

\[
T(n) = O(n^2)
\]
Outline

- Quadratic-Time Sorting
  - Bubble/Shaker Sort
  - Insertion Sort
  - Odd-Even Sort
- Linearithmic-Time Sorting
  - Heap Sort
  - Merge Sort
  - Quick Sort

Conclusion
## Summary of Sorting Algorithms

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Best</th>
<th>Average</th>
<th>Worst</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bubble</td>
<td>$O(n)$</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Shaker</td>
<td>$O(n)$</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>OddEven</td>
<td>$O(n)$</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Insertion</td>
<td>$O(n)$</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Merge</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td>Heap</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td>Quick</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>$O(n^2)$</td>
</tr>
</tbody>
</table>
In Practice...

- The only quadratic-time algorithms in use is probably insertion sort
  - Good for semi-sorted data
  - Good when $n$ is relatively small
  - Doesn't require auxiliary space

- Otherwise it's a toss-up: QuickSort vs. MergeSort
  - QuickSort doesn't require auxiliary space; merge sort does
  - MergeSort doesn't have a quadratic-time worst case
Administrivia 4/20

- Solutions posted
  - Hwk 5: Expression Trees
  - Lab 11: HashMap

- Next Tuesday is a work day
  - Attendance still required
Administrivia 4/25

- Review guide 3 posted
  - Solution will be posted Monday

- Small Hwk 6 correction

- Previously...
  - Quadratic-time sorts (sometimes can perform in linear time)
  - Today: MergeSort (linearithmic time)
Administrivia 4/27

- Sorting notes posted
- Review Monday
  - Look over review guide 3
  - Solution will be posted Monday

Previously...
- We learned about MergeSort and why it's fast
  - Eats up lots of auxiliary space though... (can we do better on space?)
- Today: Yes we can. QuickSort