Announcements

1. Midterm review options:
   1. Review sheet coming soon!
   2. Review sessions:
      1. Thursday 5-6:20pm (Shari Haynes, in B00 lecture Pepper Canyon 109)
      2. Friday 10-10:50am (me, in this room)
Today’s topics

- Selection sort and Insertion sort
- Recursion review... moving toward Quicksort and Mergesort
Sorting Algorithms
the *fruit flies* of complexity theory... (Big-O)

Checksort
check permutations until sorted

List of length *n*

| 3 | 1 | 2 |

Check:

3 1 2
3 2 1
1 2 3
1 3 2
2 1 3
2 3 1

What is the best-case running time of checksort?
A. $O(1)$
B. $O(\log(n))$
C. $O(n)$
D. $O(n \cdot \log(n))$
E. $O(n^2)$
Sorting Algorithms
the fruit flies of complexity theory... (Big-O)

Checksort
check permutations until sorted

<table>
<thead>
<tr>
<th>Check:</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 1 2</td>
</tr>
<tr>
<td>3 2 1</td>
</tr>
<tr>
<td>1 2 3</td>
</tr>
<tr>
<td>1 3 2</td>
</tr>
<tr>
<td>2 1 3</td>
</tr>
<tr>
<td>2 3 1</td>
</tr>
</tbody>
</table>

List of length n

What is the worst-case running time of checksort?
A. $O(1)$
B. $O(n \cdot \log(n))$
C. $O(n^2)$
D. $O(2^n)$
E. $O(n!)$

$O(n \cdot n!)$ comparisons for each perm.

$10 \times 9 \times 8 \times \ldots \times 2 \times 1$

$2 \times 2 \times 2 \times \ldots \times 2$

$10$
Sorting Algorithms

the *fruit flies* of complexity theory... (Big-O)

Checksort

check permutations until sorted

List of length \( n \)

```
3 1 2
```

Check:

```
3 1 2
3 2 1
1 2 3
1 3 2
2 1 3
2 3 1
```
Practical sorting algorithms: Selection sort

**Pseudocode**: selectionSort( comparableType [] array )
1. while the size of the unsorted part is greater than 1
2. find the position of the largest element in the unsorted part
3. move this largest element into the first position of the sorted part
4. decrement the size of the unsorted part by 1
   // invariant: all elements from the first position to the last position of the
   //         sorted part are in non-decreasing order

12 4 9 3 15 8 19 2

unsorted

sorted
(a) Initial configuration for selection sort. The input array is logically split into an unsorted part and a sorted part (initially empty).

(b) The array after the largest value from the unsorted part has been selected and moved to the front of the sorted part (first pass).

(c) The array after the largest value from the unsorted part has been selected and moved to the front of the sorted part (second pass).

(d) The array after the largest value from the unsorted part has been selected and moved to the front of the sorted part (third pass).
Approximately How many times does the algorithm have to select the largest element from the unsorted part of the list? (i.e. how many times does the outer loop run?)

A. 1 time
B. N times
C. $N^2$ times
Pseudocode: selectionSort( comparableType [] array )
1. while the size of the unsorted part is greater than 1
2. find the position of the largest element in the unsorted part
3. move this largest element into the first position of the sorted part
4. decrement the size of the unsorted part by 1
   // invariant: all elements from the first position to the last position of the
   //            sorted part are in non-decreasing order

- Approximately how many comparisons does it take to select the largest element from the unsorted part of the list?
  A. 1 comparison
  B. N-1 comparisons
  C. At most N-1, but usually less than N-1
Pseudocode: `selectionSort( comparableType [] array )`
1. `while the size of the unsorted part is greater than 1`
2. `find the position of the largest element in the unsorted part`
3. `move this largest element into the first position of the sorted part`
4. `decrement the size of the unsorted part by 1`
   ```
   // invariant: all elements from the first position to the last position of the
   // sorted part are in non-decreasing order
   ```

- Approximately how many comparisons does it take to select the largest element from the unsorted part of the list?
  - A. 1 comparison
  - B. N-1 comparisons
  - C. **At most N-1, but usually less than N-1**

We could overestimate use N, but we might not get the tightest bound possible.
Pseudocode: selectionSort( comparableType [] array )
1. while the size of the unsorted part is greater than 1
2. find the position of the largest element in the unsorted part
3. move this largest element into the first position of the sorted part
4. decrement the size of the unsorted part by 1
   // invariant: all elements from the first position to the last position of the
   //          sorted part are in non-decreasing order

# comparisons:
1\textsuperscript{st} iter: \( (N-1) + (N-2) + (N-3) + \ldots + 2 + 1 = S \)
2\textsuperscript{nd} iter: \( 1 + 2 + 3 + \ldots + (N-2) + (N-1) = S \)
3\textsuperscript{rd} iter: \( N + N + N + \ldots + N + N = 2S \)

\[ S = \frac{N(N-1)}{2} \Rightarrow O(N^2) \]
Insertion sort: Pseudo code

Pseudocode: insertionSort (primitiveType [] array)
1. while the size of the unsorted part is greater than 0
2. let the target element be the first element in the unsorted part
3. find target’s insertion point in the sorted part
4. insert the target in its final, sorted position
   // invariant: the elements from position 0 to (size of the sorted part – 1)
   // are in nondecreasing order

Pseudocode finding the insertion point for the target in the sorted part
1. get a copy of the first element in the unsorted part
2. while (there are elements in the unsorted part to examine AND
   we haven’t found the insertion point for the target)
3. move the element up a position // make room for the target
(a) Initial configuration for insertion sort. The input array is logically split into a sorted part (initially containing one element) and an unsorted part.

(b) The array after the first value from the unsorted part has been inserted in its proper position in the sorted part (first pass).

(c) The array after the first value from the unsorted part has been inserted in its proper position in the sorted part (second pass).

(d) The array after the first value from the unsorted part has been inserted in its proper position in the sorted part (third pass).
Insertion sort: Worst case analysis

Approximately how many times must you insert the next element into the sorted part of the array? (Outer loop)

A. 1
B. N
C. N^2
Insertion sort: Worst case analysis

- Approximately how many comparisons does it take to insert the element into the sorted part each time through the loop (in the worst case)?

A. 1
B. N
C. $N^2$
D. It depends on the length of the sorted part
Insertion sort: Worst case analysis

1st iter     2nd iter    3rd iter
# comparisons: 1 + 2 + 3 + ... + (N-2) + (N-1)

\[ \Rightarrow O(n^2) \]
Insertion sort: Best case analysis

- Approximately how many steps does it take to insert the element into the sorted part each time through the loop (in the BEST case)?

  A. 1
  B. N
  C. $N^2$
  D. It depends on the length of the sorted part

2 4 9 13 15 17 19 20