# CS240
# Fall 2014

Mike Lam, Professor

**Warm-up**
**Job Interview Question**

Choose one of the three basic sorts below and write a Python function that performs that sort on a list:

- Selection Sort
- Insertion Sort
- Bubble Sort

# Sorting

# Sorting

- “Sort” (verb)
  - “To place (records) in order, as numerical or alphabetical, based on the contents of one or more keys contained in each record.”
- Classic problem
  - Ubiquitous
  - Many approaches
  - Many minor optimizations
  - Common interview question

*"Indeed, I believe that virtually every important aspect of programming arises somewhere in the context of sorting or searching!" - Donald Knuth (1998)*

# Sorting Objectives

- From the syllabus:
  - "Implement a variety of sorting algorithms, including insertion sort, selection sort, merge sort, quicksort, and heap sort."
  - "State the asymptotic running times of the algorithms ... studied in this course, and explain the practical behavior of algorithms ...."
- More particularly:
  - Understand and articulate the sorting problem
  - Differentiate between various sort types
  - Implement examples of each sort type

# Sorting

- Best case for sorting: *O(n)*
  - Must examine/move every item
- Worst (reasonable) case for sorting: $O(n^2)$
  - Must compare every item with every other item
- There *are* worse sorts...
  - Example: Bogosort, Bozosort
  - For more info, see "Sorting the Slow Way"
- Most useful sorting algorithms are *O(n log n)* average case

# Sorting

- Algorithm evaluation criteria:
  - Best case running time
  - Worst case running time
  - Average case running time
  - Memory requirements (“space”)
  - Stability

# Sorting Stability

- If two items are equal as determined by the sort order and a given sorting algorithm will never reorder them while sorting, that sorting algorithm is *stable*
- Unstable sorts can always be modified to be stable by changing the sort order to incorporate prior order
  - `(a < b && index(a) < index(b))`
  - May require extra time or space
- Not an issue if elements are indistinguishable
- Only a problem in some domains

# Basic Sorting Algorithms

- Selection sort
  - Growing sorted region at beginning of list
  - “Select” smallest unsorted value and append to sorted region
- Insertion sort
  - Growing sorted region at beginning of list
  - “Insert” the next unsorted value into sorted region
- Bubble sort
  - Growing sorted region at end of list
  - Largest unsorted value “bubbles up” to sorted region

# Selection Sort

```
def selection_sort(items):
    """ Sort the provided Python list
        in-place using selection sort.
    """
    for j in range(len(items) - 1):
        min_index = j
        for i in range(j + 1, len(items)):
            if items[i] < items[min_index]:
                min_index = i
        tmp = items[j]
        items[j] = items[min_index]
        items[min_index] = tmp
```

# Insertion Sort

```python
def insertion_sort(items):
    """ Sort the provided Python list
        in-place using insertion sort.
    """
    for j in range(1, len(items)):
        element = items[j]
        i = j
        while 0 < i and element < items[i-1]:
            items[i] = items[i - 1]
            i -= 1
        items[i] = element
```

# Bubble Sort

```
def bubble_sort(items):
    """ Sort the provided Python list
        in-place using bubble sort.
    """
    for j in range(len(items)-1, -1, -1):
        for i in range(j):
            if items[i+1] < items[i]:
                tmp = items[i]
                items[i] = items[i+1]
                items[i+1] = tmp
```

# Bubble Sort

```
def bubble_sort(items):
    """ Sort the provided Python list
        in-place using short-circuited bubble sort.
    """
    for j in range(len(items)-1, -1, -1):
        swapped = False
        for i in range(j):
            if items[i+1] < items[i]:
                tmp = items[i]
                items[i] = items[i+1]
                items[i+1] = tmp
                swapped = True
        if not swapped:
            break
```

# Analysis

- Selection Sort $$T(n) = \sum_{1}^{n-1} i = \frac{n(n-1)}{2} \in O(n^2)$$
  - First pass does n-1 comparisons
  - Second pass does n-2 comparisons
  - etc.
- Insertion Sort $$T(n) \approx \frac{(n+4)(n-1)}{4} \in O(n^2)$$
  - Worst case is the same as selection sort
  - Suppose each inserted element is equally likely to belong at every location in the sorted region
  - Average case does roughly half the comparisons of worst case

# Basic Sorting Algorithms

- Selection sort
  - Best: $O(n^2)$ Worst: $O(n^2)$ Average: $O(n^2)$
- Insertion sort
  - Best: $O(n)$ Worst: $O(n^2)$ Average: $O(n^2)$
- Bubble sort (sort by exchange)
  - Best: $O(n)$ Worst: $O(n^2)$ Average: $O(n^2)$

(with short-circuit check; it is $O(n^2)$ otherwise)

# Basic Sorting Algorithms

- Selection sort
  - Best: $O(n^2)$ Worst: $O(n^2)$ Average: $O(n^2)$ *NOT STABLE*

$$T(n) = \sum_{1}^{n-1} i = \frac{n(n-1)}{2} \in O(n^2)$$ (every case)

(would require insertion instead of swapping to be stable)

- Insertion sort
  - Best: $O(n)$ Worst: $O(n^2)$ Average: $O(n^2)$ *STABLE*

$$T(n) = n-1 \qquad T(n) = \frac{n(n-1)}{2} \qquad T(n) \approx \frac{(n+4)(n-1)}{4}$$

- Bubble sort (sort by exchange)
  - Best: $O(n)$ Worst: $O(n^2)$ Average: $O(n^2)$ *STABLE*

(with short-circuit check; it is $O(n^2)$ otherwise)

# Shell Sort

- Generalization of insertion and bubble sort
- Concept: k-sorting
  - Starting anywhere in the list, sort every kth element
  - Repeat for successively smaller k values
  - The selected values of k are called the "gap sequence"
- Running time is hard to analyze
  - Depends on which gap sequence is chosen
  - Can be $O(n^{3/2})$ or $O(n^{4/3})$
- Not stable

# Application

- If n < 1000, any algorithm will probably do
  - Don't overcomplicate!
- If data modification is expensive, selection sort could be a good option (fewest actual writes)
- If timing predictability is necessary, use selection sort
- If the list is nearly sorted, use insertion sort
- If stability is important, avoid selection and shell sorts
- None of these run in $O(n \log n)$
  - How do we achieve this?

# Divide and Conquer

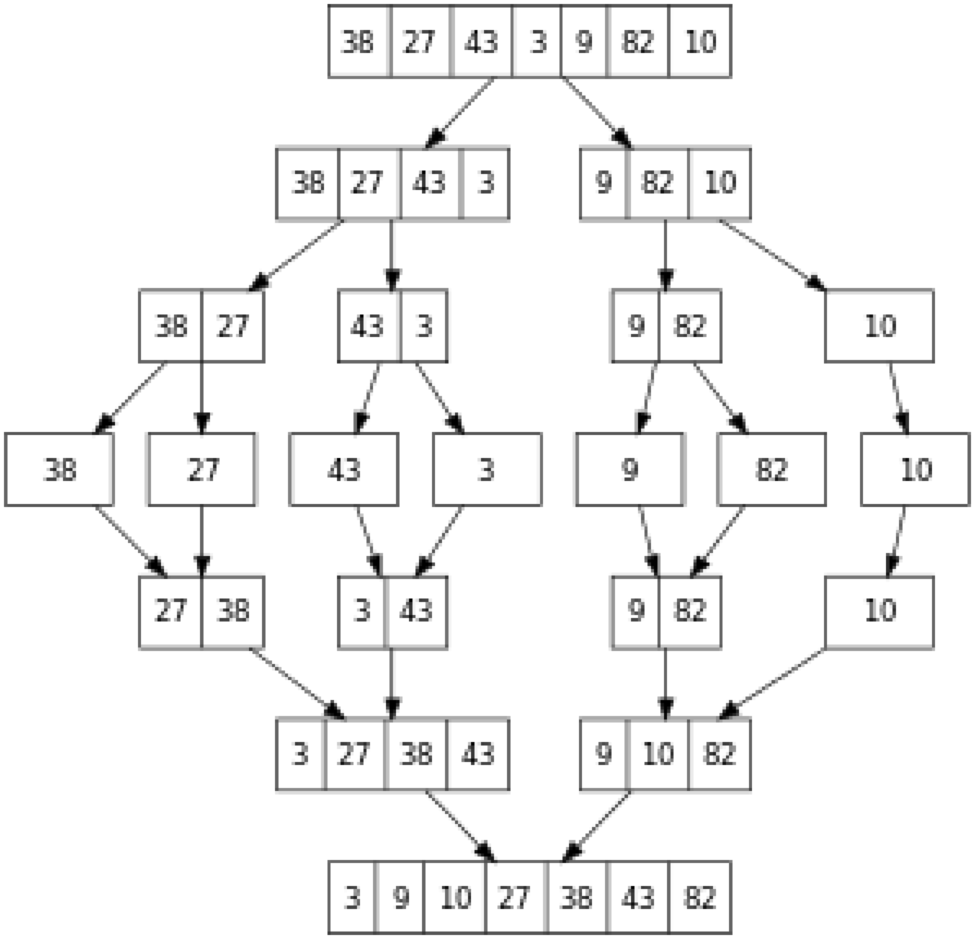
- Divide-and-conquer algorithm
  - If n < threshold, solve directly
  - Otherwise, split input into disjoint sets
    - Recursively solve subproblems
    - Combine subproblem solutions

- How could this paradigm be applied to sorting?

| | | | | | | |
|---|---|---|---|---|---|---|
| 38 | 27 | 43 | 3 | 9 | 82 | 10 |

# Merge Sort

- Visualization:



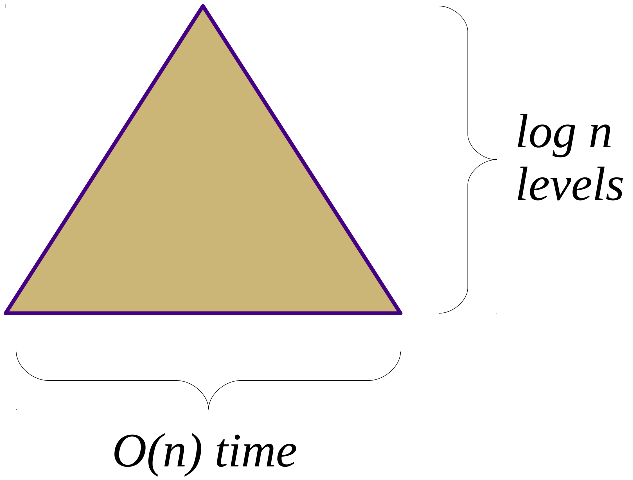

Alternate visualization
(both graphics from Wikipedia)

# Merge Sort Implementation

```python
def merge_sort(items):
    """ Sort the provided Python list using merge sort."""
    n = len(items)

    # base case: 0 or 1 items (already sorted)
    if n < 2:
        return

    # divide-and-conquer
    mid = n // 2
    left_side = items[0:mid]
    right_side = items[mid:n]
    merge_sort(left_side)       # sort left side
    merge_sort(right_side)      # sort right side

    # merge
    i = 0; j = 0
    while i + j < n:
        if not j < len(right_side) or \
                (i < len(left_side) and left_side[i] <= right_side[j]):
            items[i+j] = left_side[i]
            i += 1
        else:
            items[i+j] = right_side[j]
            j += 1
```

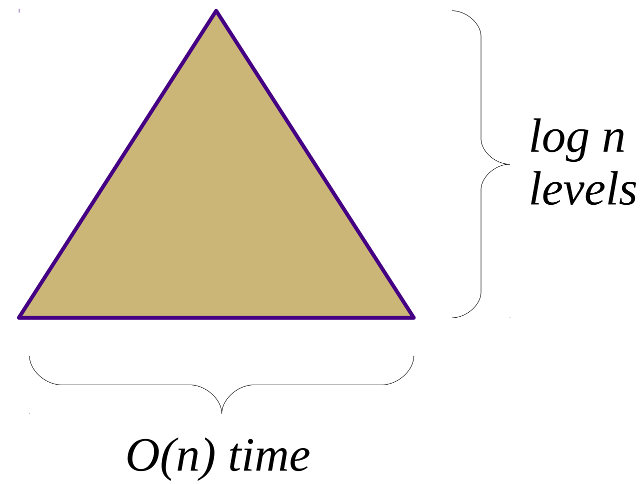
# Merge Sort Analysis

- There are $\lceil \log n \rceil$ levels of recursion
- $O(n)$ time per level
- Thus, the entire sort is $O(n \log n)$



# Merge Sort Analysis

- There are $\lceil \log n \rceil$ levels of recursion
- *O(n)* time per level
- Thus, the entire sort is *O(n log n)*
- Can also solve a recurrence:




$$
\begin{aligned}
T(0) &= 1\\
T(n) &= 2T(n/2) + cn
\end{aligned}
$$

$$
\begin{aligned}
T(n) &= 2^i T(n/2^i) + i\,n \qquad i = \log n\\
T(n) &= n + n\log n\\
T(n) &\in O(n\log n)
\end{aligned}
$$

# Merge Sort

- Alternative implementations in Section 12.2.5
  - Queue-based implementation (simpler logic)
  - Non-recursive implementation (slightly faster)
- Copying arrays is expensive
  - Not worth it once *n* is relatively small
  - Optimization: just call insertion sort when *n* is small