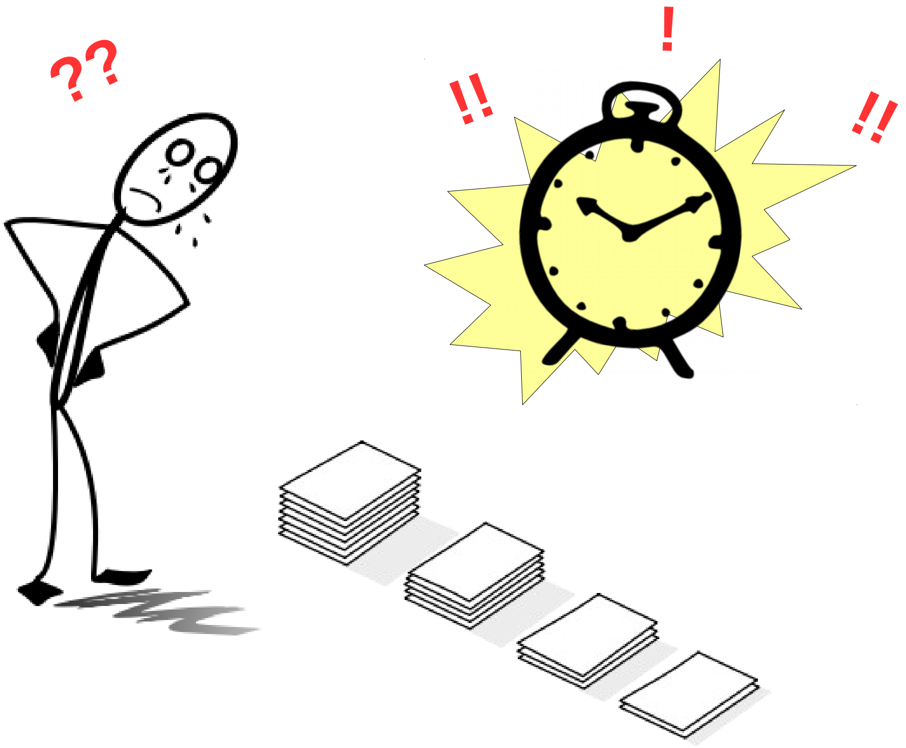
CS240
Fall 2014

Mike Lam, Professor



Quick Sort

Merge Sort

- Merge sort
  - Sort sublists (divide & conquer)
  - Merge sorted sublists (combine)
- All the "hard work" is done after recursing
- Hard to do "in-place"
  - The sublists need to be interleaved during merging
  - Doing this cleanly requires $O(n)$ extra space at minimum
- We'd like an $O(n \log n)$ algorithm that works "in-place"
  - No extra space required

Quick Sort

- Quick sort
  - Choose a pivot value
  - Partition into sublists (divide)
  - Sort sublists (conquer)
  - Merge sorted sublists (combine)
- All the "hard work" is done before recursing
  - $O(n)$ at each level
- Some work (combining) is done after recursing
  - Still $O(n)$ at each level
  - This is actually unnecessary in the in-place version

Partitioning

- Choose a *pivot* value
  - Easy choices: first, middle, or last
  - More complicated: random, median of three
- Split list into two or three sublists
  - 1) Less than and 2) Equal or Greater than
  - 1) Less than, 2) Equal to, and 3) Greater than
  - This operation can be done in-place or with auxiliary lists
- Often implemented in a separate function
  - Like the merge() operation in merge sort

Quick Sort Implementation

```python
def quick_sort(items):
    n = len(items)

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

    # choose pivot
    pivot = items[-1]   # use last item

    # divide (a.k.a. partition)
    less = []; equal = []; greater = []
    for elem in items:
        if elem < pivot:
            less.append(elem)
        elif elem > pivot:
            greater.append(elem)
        else:
            equal.append(elem)

    # conquer (recurse)
    quick_sort(less)
    quick_sort(greater)

    # combine
    i = 0
    for elem in less:
        items[i] = elem
        i += 1
    for elem in equal:
        items[i] = elem
        i += 1
    for elem in greater:
        items[i] = elem
        i += 1
```

Quick Sort Implementation

- Good: Relatively simple and easy to understand
- Bad: Uses lots of extra lists (similar to merge sort)
- Alternative: "in-place" implementation
  - Instead of building new lists during partitioning, use swapping to re-arrange sublists in the original list
  - This can be a little difficult to get exactly right
  - It's worth practicing

In-place Quick Sort

- **quick_sort**(items, *first=0*, *last=len(items)-1*):
  - pivot = choose_pivot()
  - pivot_index = partition(items, pivot)
  - **quick_sort**(items, first, pivot_index-1)
  - **quick_sort**(items, pivot_index+1, last)

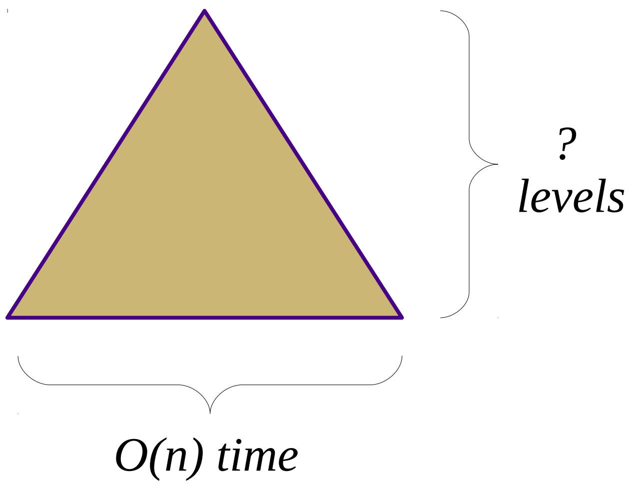
In-place Quick Sort

```python
def quick_sort_inplace(items):
    """ Sort the provided Python list using in-place quick sort."""
    quick_sort_inplace_helper(items, 0, len(items)-1)

def quick_sort_inplace_helper(items, first, last):
    """ Recursive helper for in-place quick sort."""

    # base case: 0 or 1 items (already sorted)
    if first >= last:
        return

    # choose pivot
    pivot = items[last]     # use last item

    # divide (a.k.a. partition)
    left = first
    right = last - 1    # ignore pivot for now
    while left <= right:

        # scan for values that are in the wrong partition
        # and swap them
        while left <= right and items[left] < pivot:
            left += 1
        while left <= right and pivot < items[right]:
            right -= 1
        if left <= right:
            tmp = items[left]
            items[left] = items[right]
            items[right] = tmp
            left += 1
            right -= 1

    # swap pivot with the leftmost item in the second sublist
    tmp = items[left]
    items[left] = items[last]
    items[last] = tmp

    # conquer (recurse)
    quick_sort_inplace_helper(items, first, left-1)
    quick_sort_inplace_helper(items, left+1, last)

    # no need to combine b/c we sorted in-place
```

Quick Sort Analysis

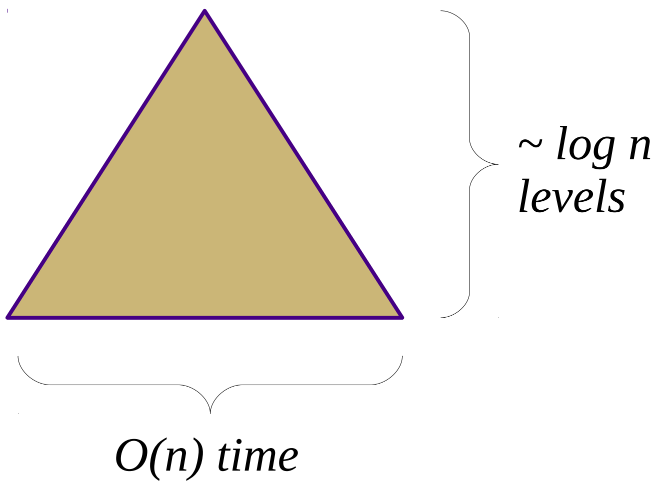
- $O(n)$ time per level
- How many levels?

(this is the key difference between analysis of merge sort and analysis of quick sort)



Quick Sort Analysis

- $O(n)$ time per level
- How many levels?
  - Best case: $\sim \log_2 n$
    - Input size is halved each time
    - Overall: $O(n \log n)$
  - Worst case: $\sim n$
    - Input size decreases by $O(1)$ each time
    - Overall: $O(n^2)$
  - Average/expected: $\sim \log_{4/3} n$
    - Equally likely to choose "good" or "bad" pivot
    - Asymptotically same as best case
    - Overall: $O(n \log n)$



Pivots

- Choice of pivot is important!
  - Determines the size of the two sublists
    - And therefore (indirectly) the recursion depth
  - Optimal: median of all values in the list
    - Sublists will be of equal length
    - Guarantees $O(n \log n)$ sort (just like merge sort)
    - Chicken-and-egg problem: calculating the median requires the list to be sorted!

Pivots

- Choice of pivot is important!
  - Non-optimal: deterministic selection
    - Choose first item, middle item, or last item
    - Picking the last item simplifies some implementations
    - Picking the middle item works well for nearly-sorted lists
    - All three have pathological cases that are $O(n^2)$
      - Picking the first or last is particularly problematic because the pathological case is a sorted list!
  - Better options: random or median-of-three
    - Randomized guarantees $O(n \log n)$ with high probability
    - Median-of-three is cheaper to compute and is similar in practice

Stability

- Quick sort is not stable
  - Partition re-orders items within sublists
- Stable variant requires O(n) extra space
  - This erases the largest advantage of quick sort over merge sort

Conclusions

- Quick sort is often the fastest comparative sort **in practice**
  - Expected $O(n \log n)$ running time in most cases
  - Requires no extra space
    - Except for $\log n$ stack frames for recursion
  - Watch out for pathological cases!
  - Many common tweaks to improve quick sort
    - Median-of-three pivot selection
    - Switch to a different sort for pathological cases
    - Switch to a different sort when n is small