

## **Week 03**

Mergesort - top-down, bottom-up,  
quicksort- 2-way, 3-way, stability,  
order of growth

# Mergesort Guide

- **Mergesort.** Merge left, merge right, merge.
- **Merge.** Understand how to carry out the merge operation. How many compares does it use when comparing two arrays of size  $N$  in the best case? In the worst case?
- **Mergesort order of growth.** Understand how to show that the order of growth of the number of compares is  $N \lg N$ . Understand why the entire algorithm is also order  $N \lg N$ .
- **Mergesort compare bounding.** Know why the best case is  $\sim \frac{1}{2} N \lg N$  and the worst case is  $\sim N \lg N$  compares.
- **Mergesort properties.** Mergesort is stable (why?). Mergesort uses  $N$  extra memory (why?). Does mergesort do particularly well on already sorted arrays? Partially ordered arrays?

# Mergesort Basics

- Divide and Conquer
  - Key idea: two sorted arrays of size  $N/2$  can be merged in linear time to get one sorted array of size  $N$
- Top-down mergesort
  - split (recursion) and merge (iteration)
- Understand how it works
- Work out a simple example
  - [5,4,2,1,3,6,5,3]
- How many compares? How many merge operations?  
For an array of size  $N$ ?

# The top-down code

```
private static void merge(Comparable[] a, Comparable[] aux, int lo, int mid, int hi)
{
    for (int k = lo; k <= hi; k++)
        aux[k] = a[k];
    // copy

    int i = lo, j = mid+1;
    for (int k = lo; k <= hi; k++)
    {
        if (i > mid)
            a[k] = aux[j++];
        else if (j > hi)
            a[k] = aux[i++];
        else if (less(aux[j], aux[i]))
            a[k] = aux[j++];
        else
            a[k] = aux[i++];
    }
    // merge
}
```

## Recursive Mergesort method

```
private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi)
{
    if (hi <= lo) return;
    int mid = lo + (hi - lo) / 2;
    sort(a, aux, lo, mid);
    sort(a, aux, mid+1, hi);
    merge(a, aux, lo, mid, hi);
}
```

Real  
work  
gets  
done  
in  
merge

# Order of Growth

- Understand how to show that the order of growth of the number of compares is  $N \lg N$ . Understand why the entire algorithm is also order  $N \lg N$

## Sorting summary

	inplace?	stable?	best	average	worst	remarks
selection	✓		$\frac{1}{2} N^2$	$\frac{1}{2} N^2$	$\frac{1}{2} N^2$	$N$ exchanges
insertion	✓	✓	$N$	$\frac{1}{2} N^2$	$\frac{1}{2} N^2$	use for small $N$ or partially ordered
shell	✓		$N \log_2 N$	?	$c N^{3/2}$	tight code; subquadratic
merge		✓	$\frac{1}{2} N \lg N$	$N \lg N$	$N \lg N$	$N \log N$ guarantee; stable
timsort		✓	$N$	$N \lg N$	$N \lg N$	improves mergesort when preexisting order
?	✓	✓	$N$	$N \lg N$	$N \lg N$	holy sorting grail

# Bounding Compares

- $\sim \frac{1}{2} N \lg N \leq \text{number of compares} \leq \sim N \lg N$  compares
- Best case:  $\frac{1}{2} N \log N$ , worst case:  $N \log N$
- What input provides a best case performance?
- What input provides a worst case performance?

# Things to know

- Mergesort is stable (why?).
- Mergesort uses  $N$  extra memory (why?).
- Does mergesort do particularly well on already sorted arrays?
- Partially ordered arrays?

# Quicksort Guide

- **Quicksort.** Partition on some pivot. Quicksort to the left of the pivot. Quicksort to the right.
- **Partitioning.** Understand exactly how to carry out 2-way partitioning as discussed in class. Be able to recognize Dijkstra's 3-way partitioning as discussed in the book.
- **Quicksort order of growth.** Understand how to show that in the best case, quicksort is  $N \lg N$ , and in the worse case is  $N^2$ . Shuffling is needed to probabilistically guarantee  $2 N \ln N$  behavior.
- **Quicksort compare counting.** Know why the best case is  $\sim N \lg N$  compares and worst case is  $\sim 1/2 N^2$ . Despite the greater number of compares, quicksort is usually faster than mergesort. Be familiar with the fact that shuffling yields  $2N \ln N$  compares on average (but you don't need to fully digest this proof -- especially solution of the difficult recurrence relation, as that involves discrete math that is beyond the scope of the course).
- **Pivot choice.** Understand how the pivot affects the size of the subproblems created after partitioning.
- **Quicksort properties.** Quicksort is not stable but it is in-place (uses no more than  $\log N$  memory).
- **Practical improvements.** Cutoff to insertion sort. Using 3-way partitioning to attain faster performance for arrays with a constant number of keys.



# Quick sort code

- **Partition** on some pivot. Quicksort to the left of the pivot. Quicksort to the right.
- Do an example:[3,5,1,2,6,4,5]

```
private static int partition(Comparable[] a, int lo, int hi)
{
    int i = lo, j = hi+1;
    while (true)
    {
        while (less(a[++i], a[lo]))           find item on left to swap
            if (i == hi) break;

        while (less(a[lo], a[--j]))           find item on right to swap
            if (j == lo) break;

        if (i >= j) break;                     check if pointers cross
        exch(a, i, j);                         swap
    }

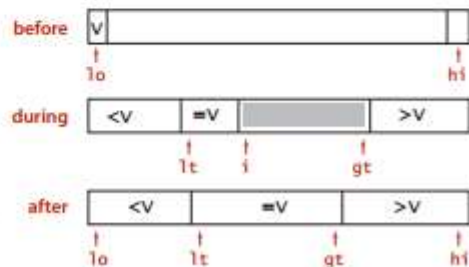
    exch(a, lo, j);                           swap with partitioning item
    return j;                                 return index of item now known to be in place
}
```

# 3-way partitioning

- Be able to recognize Dijkstra's 3-way partitioning as discussed in the book.
- Apply 3-way quicksort to
  - [5,4,2,1,3,6,5,3]

```
private static void sort(Comparable[] a, int lo, int hi)
{
    if (hi <= lo) return;
    int lt = lo, gt = hi;
    Comparable v = a[lo];
    int i = lo;
    while (i <= gt)
    {
        int cmp = a[i].compareTo(v);
        if (cmp < 0) exch(a, lt++, i++);
        else if (cmp > 0) exch(a, i, gt--);
        else i++;
    }

    sort(a, lo, lt - 1);
    sort(a, gt + 1, hi);
}
```



**2-way sort is bad with Duplicate keys. Use 3-way partitioning instead**

# Order of Growth

- Understand how to show that in the best case, quicksort is  $N \lg N$ 
  - Provide an input that gives the best case
- and in the worse case is  $N^2$ 
  - Provide an input that results in worst case
- Shuffling is needed to probabilistically guarantee  $2 N \ln N$  behavior.
  - No need to know the proof, but look at the slides

# Quicksort compare counting

- Know why the *best case* is  $\sim N \lg N$  compares and *worst case* is  $\sim 1/2 N^2$ .
- Despite the greater number of compares, *quicksort is usually faster* than mergesort.
- Be familiar with the fact that shuffling yields  $2N \ln N$  compares on average (but you don't need to fully digest this proof -- especially solution of the difficult recurrence relation, as that involves discrete math that is beyond the scope of the course).

# Pivot Choice

- **Pivot choice.** Understand how the pivot affects the size of the sub problems created after partitioning.
- What if the pivot is
  - Median of the three, first element, middle element, last element?

# Properties of Quicksort

- Quicksort is not stable but it is in-place (uses no more than  $\log N$  memory).

# Cut off to Insertion sort

- **Practical improvements.** Cutoff to insertion sort. Using 3-way partitioning to attain faster performance for arrays with a constant number of keys.

```
private static void sort(Comparable[] a, int lo, int hi)
{
    if (hi <= lo + CUTOFF - 1)
    {
        Insertion.sort(a, lo, hi);
        return;
    }
    int j = partition(a, lo, hi);
    sort(a, lo, j-1);
    sort(a, j+1, hi);
}
```