# CS/COE 1501

**www.cs.pitt.edu/~lipschultz/cs1501/**

Sorting

# The sorting problem

- Given a list of *n* items, place the items in a given order
    - Ascending or descending
        - Numerical
        - Alphabetical
        - etc.
- First, we'll review sort algorithms that fit into 3 classes:
    - Good
    - Bad
    - Ugly

# Prerequisites

```
boolean less(Comparable v, Comparable w) {
    return (v.compareTo(w) < 0);
}

void exch(Object[] a, int i, int j) {
    Object swap = a[i];
    a[i] = a[j];
    a[j] = swap;
}
```

# Bubble sort

- Simply go through the array comparing pairs of items, swap them if they are out of order
  - Repeat until you make it through the array with 0 swaps

```
void bubbleSort(Comparable[] a) {
    boolean swapped;
    do {
        swapped = false;
        for(int j = 1; j < a.length; j++) {
            if (less(a[j], a[j-1]))
                { exch(a, j-1, j); swapped = true; }
        }
    } while(swapped);
}
```

# Bubble sort example

SWAPPED!

| | | | | |
|---|---|---|---|---|
| 1 | 3 | 4 | 5 | 10 |

# “Improved” bubble sort

```
void bubbleSort(Comparable[] a) {
  boolean swapped;
  int to_sort = a.length;
  do {
    swapped = false;
    for(int j = 1; j < to_sort; j++) {
      if (less(a[j], a[j-1]))
        { exch(a, j-1, j); swapped = true; }
    }
    to_sort--;
  } while(swapped);
}
```

# How bad is it?

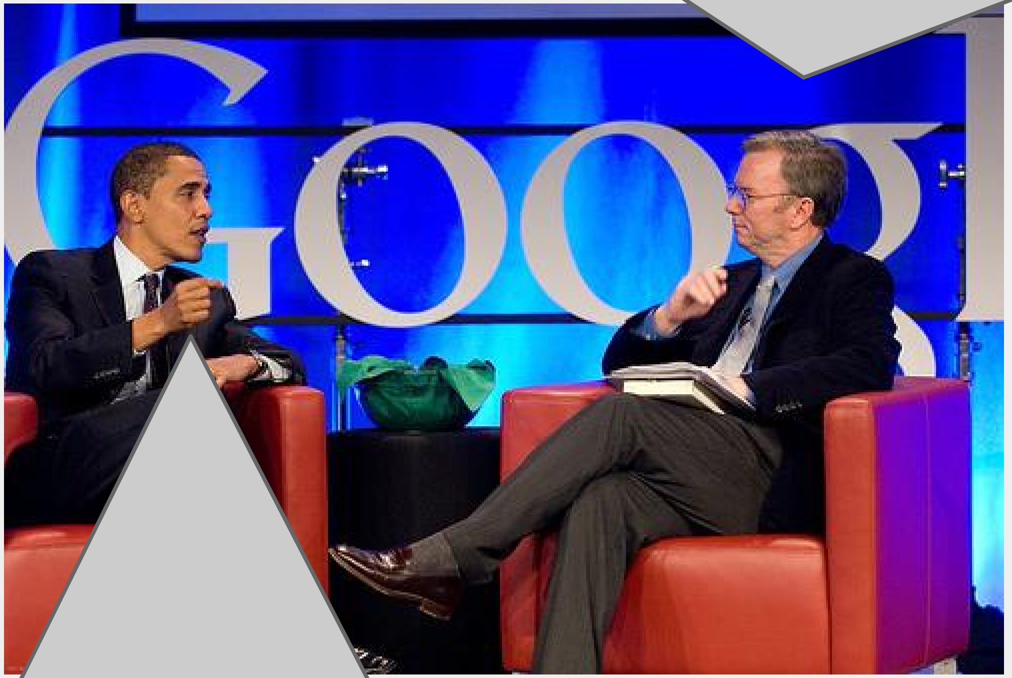
- Runtime:
  - $O(n^2)$

"[A]lthough the techniques used in the calculations [to analyze the bubble sort] are instructive, the results are disappointing since they tell us that the bubble sort isn't really very good at all."

Donald Knuth
*The Art of Computer Programming*

# The Ugly - Bubble Sort

What is the most efficient way to sort a million 32-bit integers?



I think the bubble sort would be the wrong way to go.

# The Bad - Insertion Sort

- Look at each item in the array and push it as close the front as it should go

```
void insertionSort(Comparable[] a) {
    int n = a.length;
    for (int i=1; i<n; i++) {
        for (int j=i; j>0 && less(a[j], a[j-1]); j--) {
            exch(a, j, j-1);
        }
    }
}
```

# Insertion sort example

i = 4, placing 10

| | | | | |
|---|---|---|---|---|
| 1 | 3 | 4 | 5 | 10 |

# Insertion sort model

```
void insertionSort(Comparable[] a) {
    int n = a.length;
    for (int i=1; i<n; i++) {
        for (int j=i; j>0 && less(a[j], a[j-1]); j--) {
            exch(a, j, j-1);
        }
    }
}
```

# Insertion sort analysis

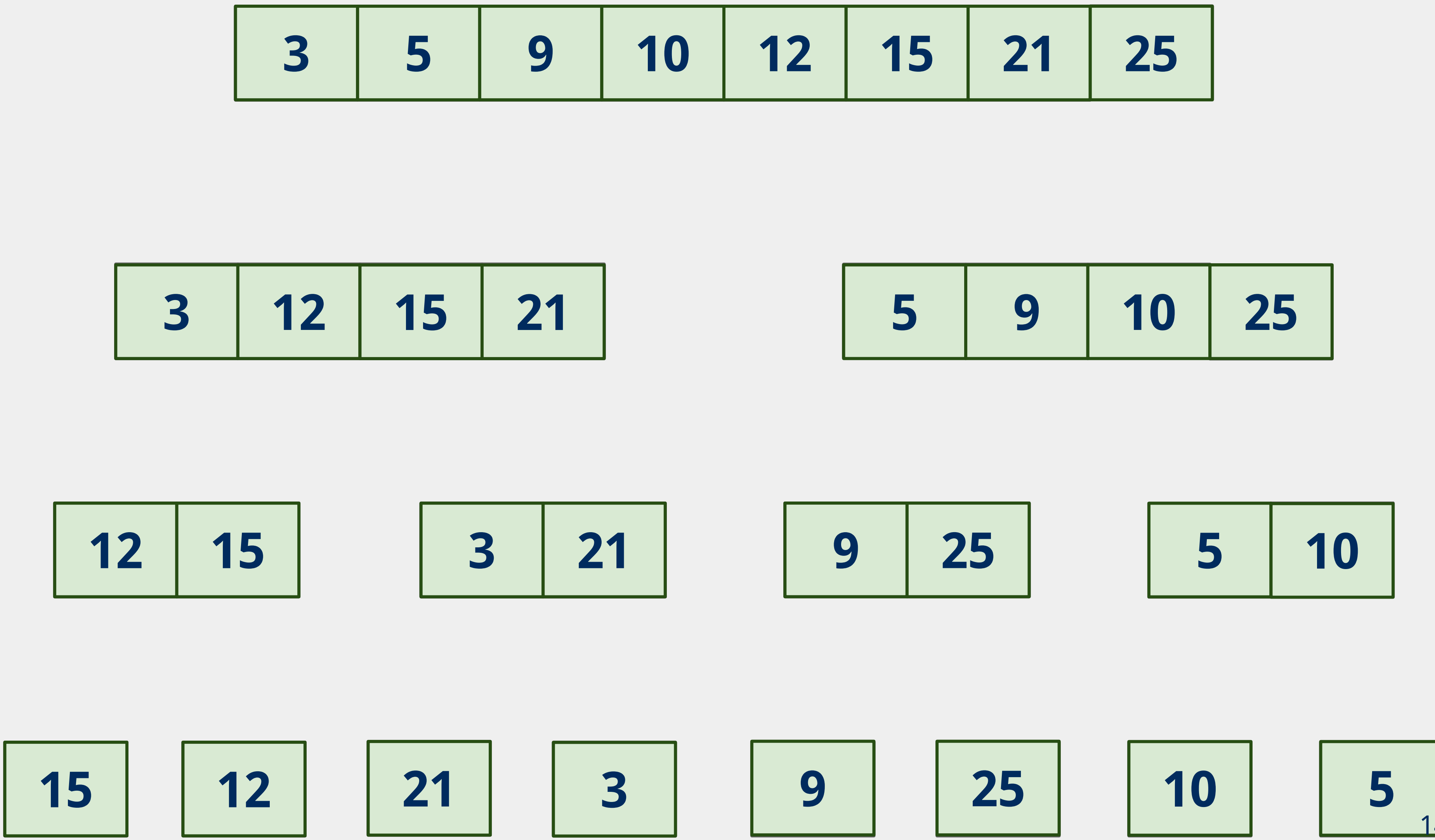
- Runtime:
  - $O(n^2)$
    - … in the worst case
  - Average case?
    - $O(n^2)$
- So why was bubble sort “Ugly”?
  - Practically, insertion sort will perform better

# The Good - Merge Sort

- Divide and conquer

```
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);
}
```

# Merge Sort trace



# Merging

```
void
merge(Comparable[] a, Comparable[] aux, int lo, int mid, int hi) {
    for (int k = lo; k <= hi; k++) {
        aux[k] = a[k];
    }
    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 sort analysis

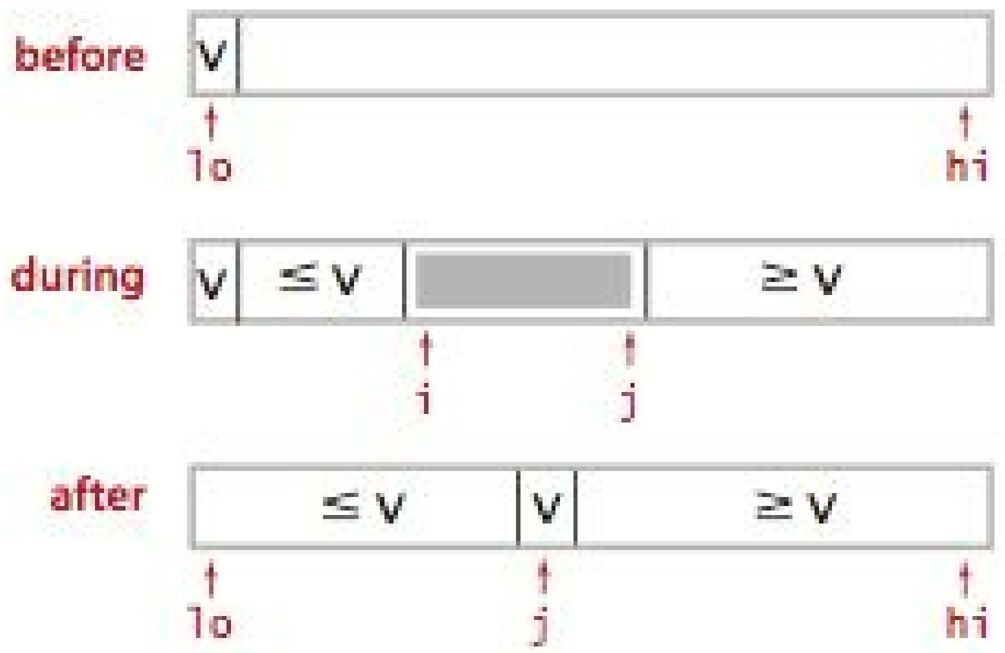
- Runtime:
  - O(n log n)
- So what's the catch?
  - Now we need O(n) space available for the aux array
    - Sort does not occur *in-place*

# The Good - Quick Sort

- Choose a *pivot* value
- Place the pivot in the array such that all items at lower indices are less than pivot, and all higher indices are greater
- Recurse for lesser indices and greater indices

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

# The Good - Quick Sort

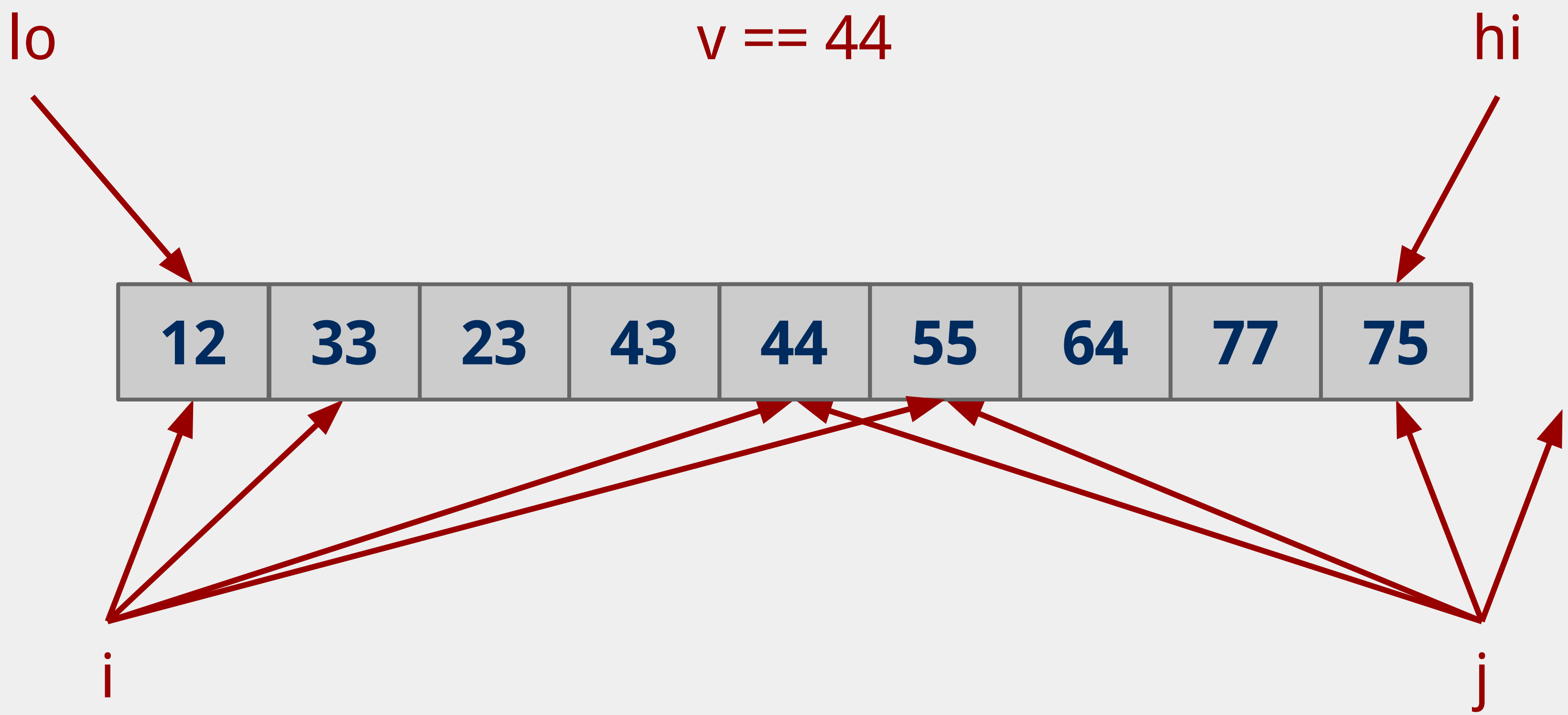


Quicksort partitioning overview

# Partitioning for quick sort

```
int partition(Comparable[] a, int lo, int hi) {
    int i = lo, j = hi + 1;
    Comparable v = a[lo];
    while (true) {
        while (less(a[++i], v))
            if (i == hi) break;
        while (less(v, a[--j]))
            if (j == lo) break;
        if (i >= j) break;
        exch(a, i, j);
    }
    exch(a, lo, j);
    return j;
}
```

# Partitioning example



# Quick sort analysis

- Runtime?
- In-place?

# This implementation of quick sort is not stable

- *Stable* sorting maintains the relative ordering of tied values

| RAM | Speed | Type | CAS ▲ | Modules | Size | Price/GB | Rating | Combo | Prime | Price |
|---|---|---|---|---|---|---|---|---|---|---|
| Corsair Dominator Platinum | DDR3-1600 | 240-pin DIMM | 7 | 2x8GB | 16GB | $14.37 | (3) | | Prime | $229.99 |
| G.Skill Trident X | DDR3-1600 | 240-pin DIMM | 7 | 2x8GB | 16GB | $10.31 | (12) | | | $164.99 |
| Mushkin Redline | DDR3-1600 | 240-pin DIMM | 7 | 2x8GB | 16GB | $11.44 | (0) | | | $182.99 |
| Mushkin Redline | DDR3-1600 | 240-pin DIMM | 7 | 2x8GB | 16GB | $11.87 | (0) | | | $189.99 |
| Crucial Ballistix | DDR3-1600 | 240-pin DIMM | 8 | 2x8GB | 16GB | $10.31 | (0) | | | $164.99 |
| Crucial Ballistix | DDR3-1600 | 240-pin DIMM | 8 | 2x8GB | 16GB | $10.31 | (15) | | Prime | $164.99 |
| Crucial Ballistix Tactical | DDR3-1600 | 240-pin DIMM | 8 | 2x8GB | 16GB | $10.00 | (13) | | Prime | $159.99 |
| Mushkin Redline | DDR3-1600 | 240-pin DIMM | 8 | 2x8GB | 16GB | $10.62 | (0) | | | $169.99 |
| Mushkin Redline | DDR3-1600 | 240-pin DIMM | 8 | 2x8GB | 16GB | $10.19 | (0) | | | $162.99 |
| A-Data XPG V1.0 | DDR3-1600 | 240-pin DIMM | 9 | 2x8GB | 16GB | $9.69 | (0) | | Prime | $154.99 |
| A-Data XPG V1.0 | DDR3-1600 | 240-pin DIMM | 9 | 2x8GB | 16GB | $9.37 | (0) | COMBO | | $149.99 |
| A-Data XPG V2 | DDR3-1600 | 240-pin DIMM | 9 | 2x8GB | 16GB | $9.69 | (2) | | | $154.99 |
| A-Data XPG V2 | DDR3-1600 | 240-pin DIMM | 9 | 2x8GB | 16GB | $9.69 | (2) | | Prime | $154.99 |
| AMD Entertainment Edition | DDR3-1600 | 240-pin DIMM | 9 | 2x8GB | 16GB | $10.12 | (0) | | Prime | $161.99 |

# Comparison sort runtime of O(n log n) is optimal

- The *problem* of sorting cannot be solved using comparisons with less than n log n time complexity
- See Proposition I in Chapter 2.2 of the text

# How can we sort without comparison?

- Consider the following approach:
  - Look at the least-significant digit
  - Group numbers with the same digit
    - Maintain relative order
  - Place groups back in array together
    - I.e., all the 0’s, all the 1’s, all the 2’s, etc.
  - Repeat for increasingly significant digits

# Radix sort analysis

- Runtime?
- In-place?
- Stable?

# Further thoughts on Eric Schmidt's question...

- 1,000,000 32-bit integers don't take up a whole lot of space
  - 4 MB
- What if we needed to sort 1TB of numbers?
  - Won't all fit in memory…
  - We had been assuming we were performing *internal* sorts
    - Everything in memory
  - We now need to consider *external* sorting
    - Where we need to write to disk

# Hybrid merge sort

- Read in amount of data that will fit in memory
- Sort it in place
  - I.e., via quick sort
- Write sorted chunk of data to disk
- Repeat until all data is stored in sorted chunks
- Merge chunks together

# External sort considerations

- Should we merge all chunks together at once?
  - Means fewer disk read/writes
    - Each merge pass reads/writes every value
  - But also more disk seeks
- Can we do parallel reads/writes to multiple disks?
- Can we use multiple CPUs/cores to speed up processing

# Large scale sorts

- What about when you have 1PB of data?
- In 2008, Google sorted 10 trillion 100 byte records on 4000 computers in 6 hours 2 minutes
- 48,000 hard drives were involved
  - At least 1 disk failed during each run of the sort