CSCI 136
Data Structures & Advanced Programming

Faster Sorting Methods

Goals

Introduce *Divide & Conquer* algorithm design

Explore two efficient D&C sorting methods

- MergeSort
- QuickSort

Divide & Conquer

Binary search is efficient because

- Divides the data in half (in constant time)
- Eliminates[1] one of the halves (in constant time)

So, the number of value comparisons T_n needed to search an array of size n satisfies

$$T_n = T_{\frac{n}{2}} + 1 \text{ and } T_1 = 1$$

By induction, we can show that T_n is $O(\log_2 n)$

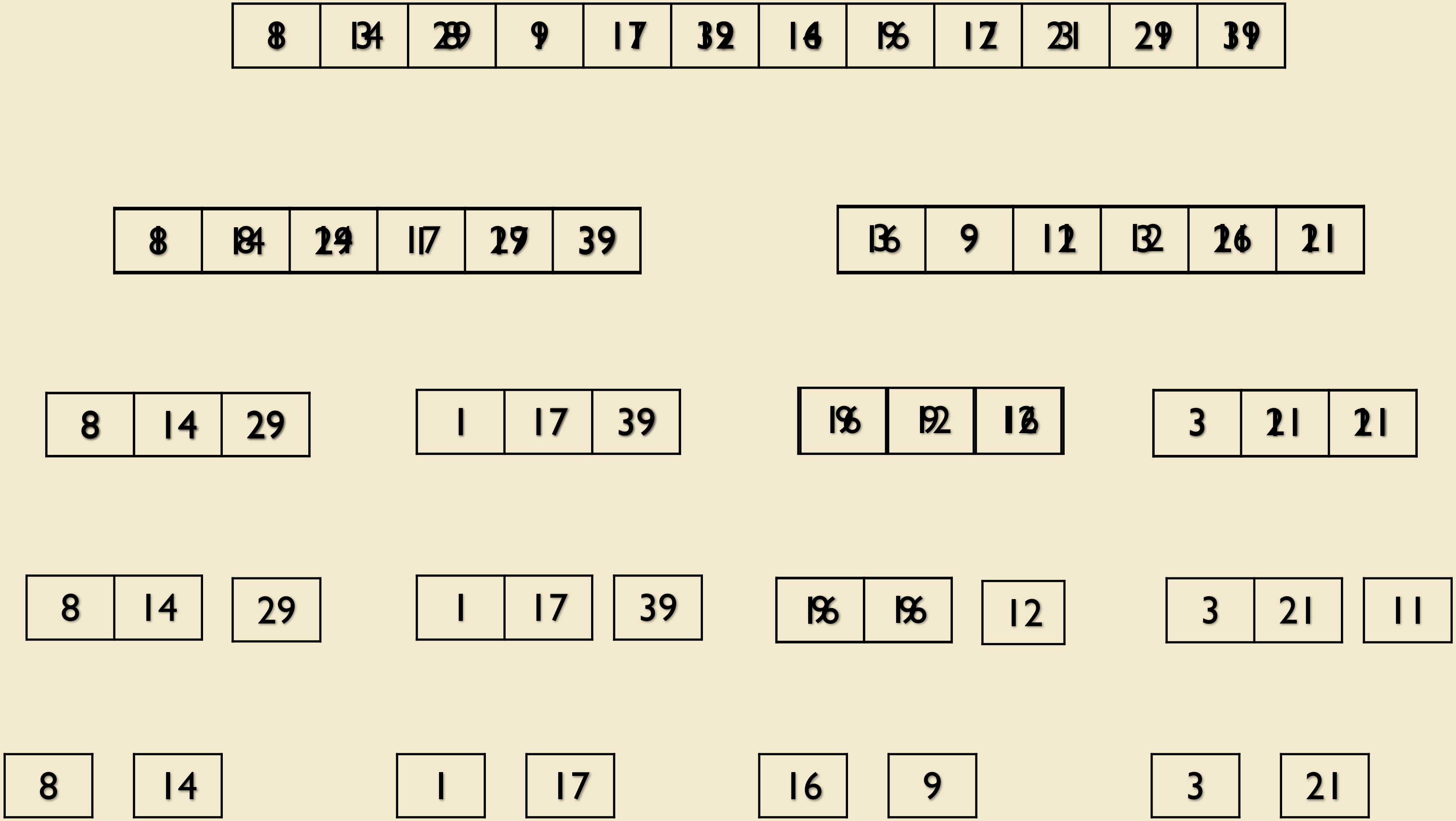
- Try this at home! [Hint: Prove it's $O(1 + \log_2 n)$]
  - You'll need to use *strong* induction

[1] :The "conquer" step

Faster Sorting: Merge Sort

- A *divide and conquer* algorithm
- Typically used on arrays
- Merge sort works as follows:
  - If the array is of length 0 or 1, then it is already sorted.
  - Divide the unsorted array into two arrays of about half the size of original.
  - Sort smaller arrays recursively by re-applying merge sort.
  - Merge the two smaller arrays back into one sorted array.
- Time Complexity?
  - Spoiler Alert! We'll see that it's O(n log n)
- Space Complexity?
  - O(n)

Merge Sort



Merge Sort : Pseudo-code

- How would we *design* it?
- First pass…

```
// recursively mergesorts A[from .. To] "in place"
void recMergeSortHelper(A[],  int from, int to)
    if ( from ≤ to )
        mid = (from + to)/2
        recMergeSortHelper(A, from, mid)
        recMergeSortHelper(A, mid+1, to)
        merge(A, from, to)
```

But *merge* hides a number of important details….

Merge Sort : An Optimization

A naive merge method creates a secondary array

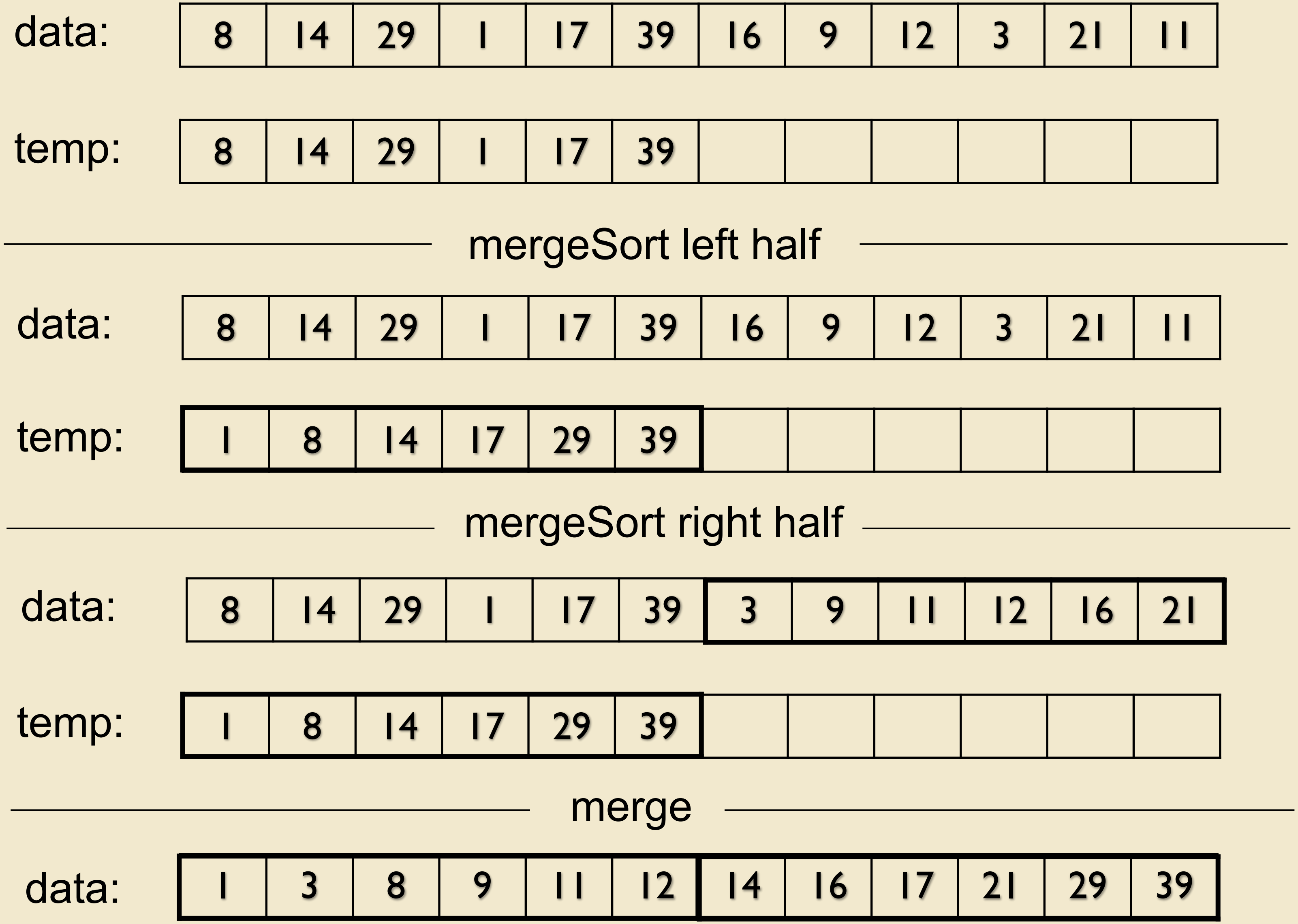
- The two merged halves of the original array are merged into the secondary array
- The secondary array is copied back into the original array

This involves lot of array creation and moving

Instead, merge

- Uses a *single* secondary array
- Merges left half of original array to secondary

Merge Sort : Temporary Array



data:	8	14	29	1	17	39	16	9	12	3	21	11
temp:	8	14	29	1	17	39						

mergeSort left half

data:	8	14	29	1	17	39	16	9	12	3	21	11
temp:	1	8	14	17	29	39						

mergeSort right half

data:	8	14	29	1	17	39	3	9	11	12	16	21
temp:	1	8	14	17	29	39						

merge

data:	1	3	8	9	11	12	14	16	17	21	29	39

Merge Sort Implementation

```
private static <T extends Comparable<T>> void
       mergeSortRecursive(T[] data, T[] temp, int low, int high) {

   int n = high-low+1;
   int middle = low + n/2;
   if (n < 2) return;

   int i;
   // move lower half of data into temporary storage
   for (i = low; i < middle; i++) {
        temp[i] = data[i];
   }
   mergeSortRecursive(temp,data,low,middle-1);
   mergeSortRecursive(data,temp,middle,high);
   merge(data,temp,low,middle,high);
}
```

Merge Method

```
private static <T extends Comparable<T>> void
       merge(T[] dest, T[] using, int low, int middle, int high) {

   int ri = low; // result index
   int ui = low; // using index
   int di = middle; // dest index
   // while two lists are not empty merge smaller value
   while (ui < middle && di <= high) {
        if (dest[di].compareTo(using[ui]) < 0)
                dest[ri++] = dest[di++]; // smaller is in high dest
        else
                dest[ri++] = using[ui++]; // smaller is in using
   }
   // possibly some values left in using array
   while (ui < middle) {
         dest[ri++] = using[ui++];
   }
}
```

Aside: n++ vs ++n

The *postfix* increment (++) operator adds 1 to the value of the variable to which it is applied

```
int n = 1;
n++;             // n now has value 2
int k = n++;     // k now has value 2 and n has value 3
```

Note that, in addition to incrementing n, it also returns the *pre-increment* value of n

- That is: ++ is an operator that
  - returns the value of the variable to which it is applied
  - after which it increments the value of that variable

There is also a *prefix* version: ++n

- It first increments the value of the variable
- after which it returns the (incremented) value

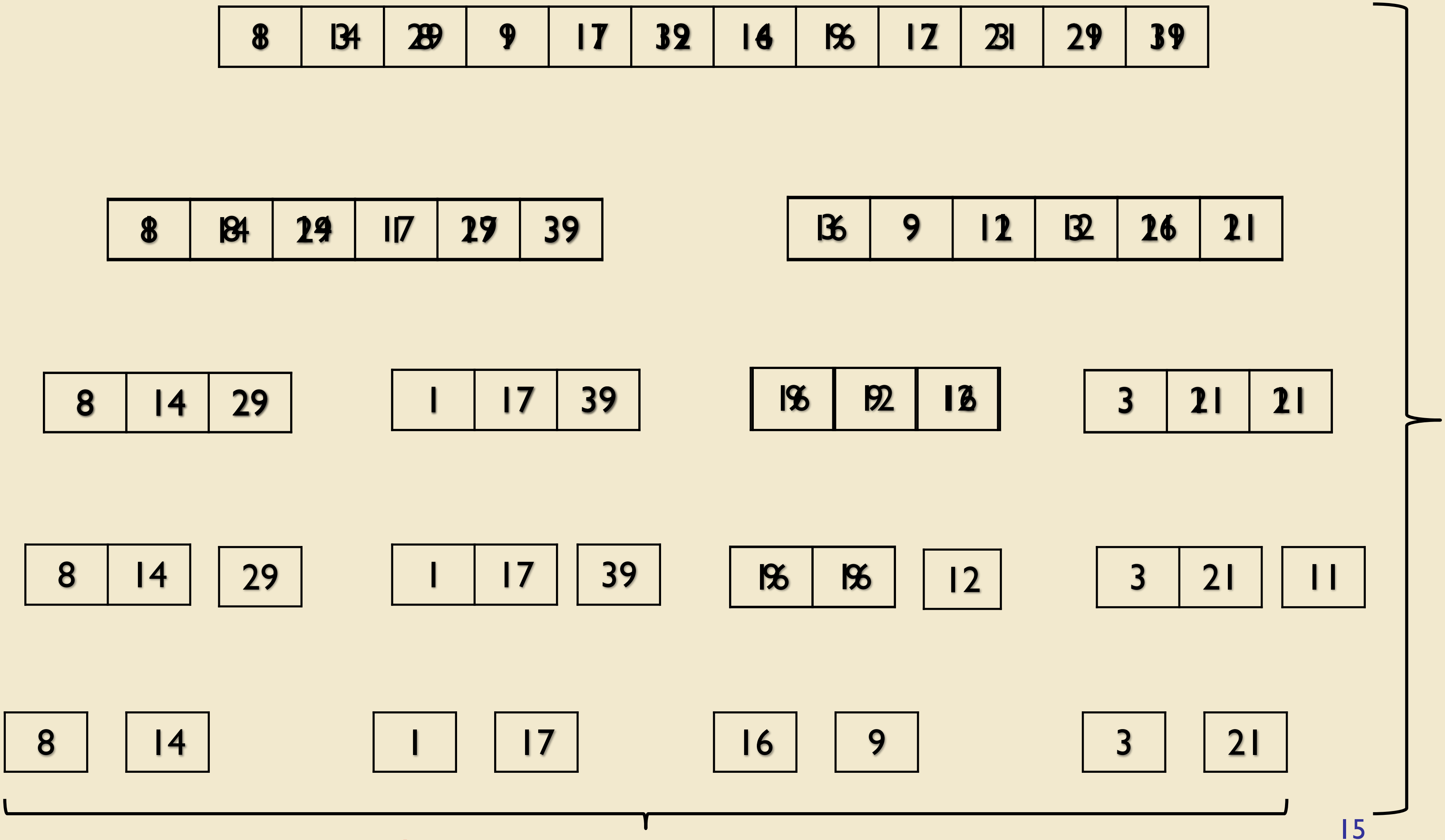
```
k = ++n;                     // k now has value 4 and so does n
```

Merge Sort : Java Implementation

- Implementation Notes
  - Note use of generics
  - Note carefully how temp array is used to reduce copying
  - Make sure the data is in the correct array!
- Time Complexity?
  - Takes at most 2k comparisons to merge two lists of size k
  - Number of splits/merges for list of size n is log n
  - Claim: At most time O(n log n)…We'll see soon...
- Space Complexity?
  - O(n)?
  - Need an extra array, so really O(2n)! But O(2n) = O(n)

Merge Sort

log(n)
depth



merge takes at most n comparisons per line

Time Complexity Proof

- Prove for $n = 2^k$ (true for other n but harder)
- That is, MergeSort for performs at most
  - $n * \log(n) = 2^k * k$ comparisions of elements
- Base cases $k \leq 1$: 0 comparisons: $0 < 1 * 2^1$ ✓
- Induction Step: Suppose true for all integers smaller than k. Let T(k) be # of comparisons for 2^k elements. Then
- $\underline{T(k) \leq} \, 2^k + 2*T(k-1) \leq 2^k + 2(k-1)2^{k-1} \leq \underline{k*2^k}$ ✓

Merge Sort

- Unlike Bubble, Insertion, and Selection sort, Merge sort is a divide and conquer algorithm
  - Bubble, Insertion, Selection sort complexity: $O(n^2)$
  - Merge sort complexity: $O(n \log n)$
- Are there any limitations with Merge sort?
  - What if we're dealing with singly-linked lists?
- Why would we ever use any other algorithm for sorting?

Merge Sort for SLL

- Finding the middle element of the list takes a linear number of steps, so does merging two sorted lists of total length n
- So, if T(n) is the number of steps necessary to mergeSort an n-element list then

$$T(n) \leq 2T\left(\frac{n}{2}\right) + cn$$

- where cn represents the combined number of steps for splitting the list and merging the two halves

Time Complexity Proof Revisited

- Claim: For $n \geq 1, T(n) \leq 2cn \log_2 n + c$
  - Base case $n \leq 1$: Let c be the number of statements in mergeSort. Then at most c statements are executed. And $c \leq 2c \log_2 1 + c$ ✓
- Induction Step: Suppose true for all integers smaller than n, for some $n > 1$. That is,
  - For all $k < n, T(k) \leq 2ck \log_2 k + c$
  - Now must show: $T(n) \leq 2cn \log_2 n + c$
- Recall: $T(n) \leq 2T\left(\frac{n}{2}\right) + cn$
  - Two recursive calls to mergeSort plus a merge

Time Complexity Proof Revisited

$$
\begin{aligned}
T(n) &\le 2T\left(\frac{n}{2}\right) + cn \\
&\le 2\left(2c\frac{n}{2}\log_2\frac{n}{2} + c\right) + cn \qquad \textit{(by induction)} \\
&\le 2cn\log_2\frac{n}{2} + 2c + cn \\
&\le 2cn\log_2 n - 2cn + 2c + cn \\
&= 2cn\log_2 n - cn + 2c \\
&\le 2cn\log_2 n + c + c(1-n) \\
&\le 2cn\log_2 n + c \text{ for } n \ge 1
\end{aligned}
$$

Drawbacks to Merge Sort

- Need extra temporary array
  - If data set is large, this could be a problem
- Waste time copying values back and forth between original array and temporary array
- Can we avoid this?

Quick Sort

- Quick sort is designed to behave much like Merge sort, without requiring extra storage space

Merge Sort	Quick Sort
Divide list in half	Partition* list into 2 parts
Sort halves	Sort parts
Merge halves	Join* sorted parts

Quick Sort

```
public void quickSortRecursive(Comparable data[],
                    int low, int high) {
    // pre: low <= high
    // post: data[low..high] in ascending order
        int pivot;
        if (low >= high) return;

       /* 1 - place pivot */
        pivot = partition(data, low, high);
       /* 2 - sort small */
       quickSortRecursive(data, low, pivot-1);
       /* 3 - sort large */
       quickSortRecursive(data, pivot+1, high);
}
```

Partition

Idea : Rearrange array so that

- First element (pivot) into its final (sorted) position
- All values smaller than pivot are to the left of pivot
- All values larger than pivot are to the right of pivot
- Return index of “pivot”

Partition : Pivot is 8

8	14	29	1	17	39	16	9	12	3	21	11
L											R

8	14	29	1	17	39	16	9	12	3	21	11
L									R		

3	14	29	1	17	39	16	9	12	8	21	11
	L								R		

3	8	29	1	17	39	16	9	12	14	21	11
	L							R			

3	8	29	1	17	39	16	9	12	14	21	11
	L		R								

3	1	29	8	17	39	16	9	12	14	21	11
		L	R								

3	1	8	29	17	39	16	9	12	14	21	11
		L=R									

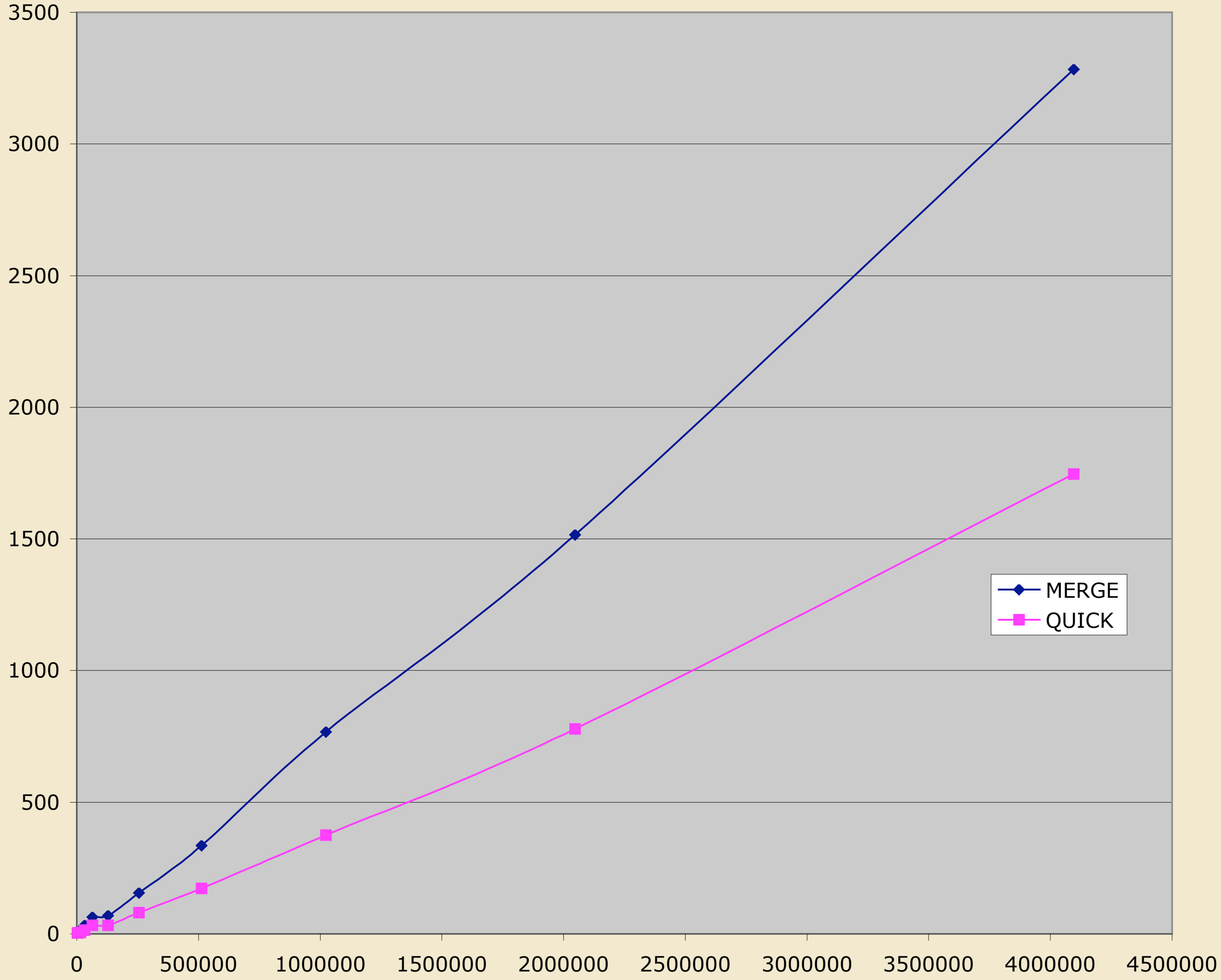
Partition

```
int partition(int data[], int left, int right) {
   while (true) {
        // pivot is data[left]: Compare to values on its right
        while (left < right && data[left] < data[right]) right--;

        if (left < right) swap(data,left++,right);
        else return left;

        // switch sides!

        // pivot is data[right]: Compare it to values on its left
        while (left < right && data[left] < data[right]) left++;

        if (left < right) swap(data,left,right--);
        else return right;
   }
}
```

Complexity

- Time:
  - Partition is O(n)
  - If partition breaks list exactly in half, same as merge sort, so O(n log n)
  - If data is already sorted, partition splits list into groups of 1 and n-1, so O(n^2)
- Space:
  - O(n) (so is MergSort)
    - In fact, it's n + c compared to 2n + c for MergeSort

Merge vs. Quick (Average Time)



Food for Thought…

- How to avoid picking a bad pivot value?
  - Pick median of 3 elements for pivot (heuristic!)
- Combine selection sort with quick sort
  - For small n, selection sort is faster
  - Switch to selection sort when elements is <= 7
  - Switch to selection/insertion sort when the list is almost sorted (partitions are very unbalanced)
    - Heuristic!

Sorting Wrapup

	Time	Space
Bubble	Worst: O(n^2) Best: O(n) - if "optimized"	O(n) : n + c
Insertion	Worst: O(n^2) Best: O(n)	O(n) : n + c
Selection	Worst = Best: O(n^2)	O(n) : n + c
Merge	Worst = Best:: O(n log n)	O(n) : 2n + c
Quick	Average = Best: O(n log n) Worst: O(n^2)	O(n) : n + c