CSCI 136
Data Structures &
Advanced Programming

Lecture 9

Fall 2019

Instructors: Bill & Sam

Administrative Details

- Remember: First Problem Set due at beginning of class on Friday
- Lab 3 Today!
  - Come to lab with a plan!
  - Answer questions before lab

Last Time

- Weak induction
- Recursion

Today's Outline

- Finish Binary Search & Strong Induction
- Recursion example: longest increasing subsequence
- Basic Sorting
  - Insertion, Selection Sorts
  - Including proofs of correctness

Example: Binary Search

- Given an array a[] of positive integers in increasing order, and an integer x, find location of x in a[].
  - Take "indexOf" approach: return −1 if x is not in a[]

```
protected static int recBinarySearch(int a[], int value,
              int low, int high) {
   if (low > high) return -1;
   else {
       int mid = (low + high) / 2;        //find midpoint
       if (a[mid] == value) return mid;  //first comparison
                                          //second comparison
       else if (a[mid] < value)           //search upper half
       return recBinarySearch(a, value, mid + 1, high);
        else                              //search lower half
              return recBinarySearch(a, value, low, mid - 1);
   }
```

Binary Search takes O(log n) Time

Can we use induction to prove this?

- Claim: If n = high - low $+1$, then recBinSearch performs at most c $(1 + \log n)$ operations, where c is *twice* the number of statements in recBinSearch
- Base case: $n = 1$: Then low = high so only c statements execute (method runs twice) and $c \leq c(1 + \log 1)$
- Assume that claim holds for some $n \geq 1$, does it hold for $n + 1$? [Note: $n + 1 > 1$, so low < high]
- Problem: Recursive call is *not* on n---it's on n/2.
- Solution: We need a better version of the PMI….

Mathematical Induction

Principle of Mathematical Induction (Strong)

Let P(0), P(1), P(2), ... be a sequence of statements (each of which could be either true or false).

Suppose that, for some $k \geq 0$

1. P(0), P(1), ... , P(k) are true, and
2. For every $n \geq k$, if P(1), P(2), ... , P(n) are true, then so is P(n+1).

Then all of the statements are true!

Binary Search takes O(log n) Time

Try again now:

- Assume that for some $n \geq 1$, the claim holds *for all* $k \leq n$, does claim hold for $n + 1$?
- Yes! Either
  - x = a[mid], so a constant number of operations are performed, or
  - RecBinSearch is called on a sub-array of size at most n/2, and by induction, at most c(1 + log (n/2)) operations are performed.
    - This gives a total of at most c + c(1 + log(n/2)) = c + c(log(2) + log(n/2)) = c + c(log n) = c(1 + log n) statements

Notes on Induction

- Whenver induction is needed, strong induction *can* be used
- The numbering of the propositions doesn't need to start at 0
- The number of base cases depends on the problem at hand
  - Enough are needed to guarantee that the smallest non-base case can be proven using only the base cases

Longest Increasing Subsequence

- Given an array a[] of positive integers, find the length of the largest subsequence of (not necessary consecutive) elements such that for any pair a[i], a[j] in the subsequence, if i<j, then a[i] < a[j].
- Example 10 7 12 3 5 11 8 9 1 15 has 3 5 8 9 15 as its longest increasing subsequence (LIS), so length is 5.
- How could we find the LIS length of a[]?
- How could we prove our method was correct?
- Let's think....

Longest Increasing Subsequence

- We'll assume all numbers are positive
- (Brilliant) Observation: A LIS for a[1 ... n] either contains a[1] … or it doesn't.
- Therefore, a LIS for a[1 ... n] either
  - Doesn't contain a[1] and is just a LIS for a[2 ... n]
  - Does contain a[1], along with an LIS for a[2 ... n] such that every element in the LIS is > a[1], or
- So the LIS length is either
  - Or the LIS length for a[2..n]
  - 1 + LIS length for a[2..n] where every element in LIS is > a[1]
- So the problem to solve is: find the LISL for a[] given that every element in LIS is at least some *threshold* value

Longest Increasing Subsequence

```
// Pre: curr < arr.length
// Post: returns length of LIS of arr[curr…] having all > threshold
public static int lisHelper(int[] arr, int curr, int threshold ) {
    if(curr == arr.length -1)
        if (return arr[curr] > threshold) return 1;
        else return 0;
    else
        int usingFirst = 0;
        if(arr[curr] > threshold)
            usingFirst  = 1 + lisHelper(arr, curr+1, arr[curr]);
        int notUsingFirst = lisHelper(arr, curr+1, threshold);
        return Math.max(usingFirst, notUsingFirst);
  }
```

Sorting Intro: Bubble Sort

- Simple sorting algorithm that works by repeatedly stepping through the list to be sorted, comparing two items at a time and swapping them if they are in the wrong order
- Repeated until no swaps are needed
- Gets its name from the way larger elements "bubble" to the end of the list
- Inefficient (both in theory and practice)
- Details in textbook

Sorting Intro: Insertion Sort

- 5 7 0 3 4 2 6 1
- 5 7 0 3 4 2 6 1
- 0 5 7 3 4 2 6 1
- 0 3 5 7 4 2 6 1
- 0 3 4 5 7 2 6 1
- 0 2 3 4 5 7 6 1
- 0 2 3 4 5 6 7 1
- 0 1 2 3 4 5 6 7

http://www.visualgo.net/sorting

Sorting Intro : Insertion Sort

- Simple sorting algorithm that works by building a sorted list one entry at a time
- Less efficient on large lists than more advanced algorithms
- Advantages:
  - Simple to implement and efficient on small lists
  - Efficient on data sets which are already mostly sorted
- Time complexity
  - $O(n^2)$
- Space complexity
  - $O(n)$

Sorting Intro : Insertion Sort

```
void iisort(int[] a, int n){
      int i,j;
      for(i=1;i<n;i++)
            for(j=i; j>0 && a[j-1] > a[j];j--)
                  swap(j, j-1, a);
}
```

Sorting Intro : Selection Sort

http://www.visualgo.net/sorting

(demo is "min" version)

- 11 3 27 5 16
- 11 3 16 5 27
- 11 3 5 16 27
- 5 3 11 16 27
- 3 5 11 16 27

- Time Complexity:
  - $O(n^2)$
- Space Complexity:
  - $O(n)$

Sorting Intro : Selection Sort

- Similar to insertion sort
- Easier to show correct (?)
- The algorithm works as follows:
  - Find the maximum value in the list
  - Swap it with the value in the last position
  - Repeat the steps above for remainder of the list (ending at the second to last position)

Some Skill Testing!

Selection sort uses two utility methods

Uses a swap method

```
private static void swap(int[]A, int i, int j) {
        int temp = a[i];
        A[i] = A[j];
        A[j] = temp;
}
```

And a max-finding method

```
// Find position of largest value in A[0 .. last]
public static int findPosOfMax(int[] A, int last) {
        int maxPos = 0;      // A wild guess
        for(int i = 1; i <= last; i++)
                if (A[maxPos] < A[i]) maxPos= i;
        return maxPos;
}
```

Some Skill Testing!

An Iterative Selection Sort

```
public static void selectionSort(int[] A) {
       for(int i = A.length - 1; i>0; i--)
           int big= findPosOfMax(A,i);
           swap(A, i, big);
       }
}
```

A Recursive Selection Sort (just the helper method)

```
public static void recSSHelper(int[] A, int last) {
       if(last == 0) return; // base case

       int big= findPosOfMax(A, last);
       swap(A,big,last);
       recSSHelper(A, last-1);
}
```

Some Skill Testing!

- Prove: recSSHelper (A, last) sorts elements A[0]…A[last].
  - Assume that maxLocation(A, last) is correct
- Proof:
  - Base case: last = 0.
  - Induction Hypothesis:
    - For k<last, recSSHelper sorts A[0]…A[k].
  - Prove for last:
    - Note: Using Second Principle of Induction (Strong)

Some Skill Testing!

- After call to findPosOfMax(A, last):
  - 'big' is location of largest A[0..last]
- That value is swapped with A[last]:
  - Rest of elements are A[0]..A[last−1].
- Since last − 1< last, then by induction
  - recSSHelper(A, last−1) sorts A[0]..A[last−1].
- Thus A[0]..A[last−1] are in increasing order
  - *and* A[last−1] ≤ A[last].
- So, A[0]···A[last] are sorted.

Making Sorting Generic

- We need *comparable* items
- Unlike with equality testing, the Object class doesn't define a "compare()" method 😟
- We want a uniform way of saying objects can be compared, so we can write generic versions of methods like binary search
- Use an interface!
- Two approaches
  - Comparable interface
  - Comparator interface

Java Interfaces : Motivating Example

- Idea: Implement a class that describes a single playing card (e.g., “Queen of Diamonds”)
- Start simple: a single class – BasicCard
- Think about alternative implementations
- Use an *interface* to allow implementation independent coding
- Let’s look at BasicCard

Aside : Enum Types are Class Types

```
enum Rank { TWO, THREE, FOUR, FIVE, SIX, SEVEN,
      EIGHT, NINE, TEN, JACK, QUEEN, KING, ACE;
}
```

Notes

- Creates an ordered sequence of named constants
- Can find position of an enum value in sequence
  - `int i = r.ordinal(); // r is of type Rank`
- Can get an array of all values in the enum
  - `Rank[] allRanks = Rank.values();`
- Can use in **for** loops
  - `for (Rank r : Rank.values() ) { ... }`
- Can have its own instance variables and methods

Implementing a Card Object

- Think before we code!
- Many ways to implement a card
  - An index from 0 to 51; a rank and a suit, ...
- Start general.
  - Build an *interface* that advertises all public features of a card
  - Not an implementation (define methods, but don't include code)
- Then get specific.
  - Build specific implementation of a card using our general card interface

Start General: Card: An Interface

- What data do we have to represent?
  - Properties of cards
  - How can we represent these properties?
    - There are often multiple options—name some!
- What methods do we need?
  - Capabilities of cards
  - Do we need *accessor* and/or *mutator* methods?

*

A Card Interface

```
public interface Card {

    // Methods - must be public
    public Suit getSuit();
    public Rank getRank();
}
```

Notes

- Don't allow card to change its value
  - Only need accessor methods
- Support `enums` for rank and suit

Get Specific: Card Implementations

- Now suppose we want to build a specific card object
- We want to use the properties/capabilities defined in our interface
  - That is, we want to *implement* the interface

```
public class CardRankSuit implements Card {
    . . .
}
```

The Enums for Cards

```
public enum Suit {
    CLUBS, DIAMONDS, HEARTS, SPADES; // the values

public String toString() {
        switch (this) {
        case CLUBS : return "clubs";
        case DIAMONDS : return "diamonds";
        case HEARTS : return "hearts";
        case SPADES : return "spades";
        }
        return "Bad suit!";
    }
}
```

A similar declaration is defined for Rank

A First Card Implementation

```
public class CardRankSuit implements Card {
// instance variables
      protected Suit suit;
      protected Rank rank;
// Constructors
      public CardRankSuit( Rank r, Suit s)
            {suit = r; rank = s;}
// returns suit of card
      public Suit getSuit() { return suit;}
// returns rank of card
      public Rank getRank() { return rank;}
// create String representation of card
 public String toString()
           {return getRank() + " of " + getSuit();}
}
```

A Second Card Implementation

```
public class Card52 implements Card {
// instance variables
protected int code; // 0 <= code < 52;
// rank is code/13 and suit is code%13
// Constructors
public CardRankSuit( int index )
      {code = index;}
// returns suit of card
      public Suit getSuit() {// see sample code}
// returns rank of card
      public Rank getRank() {// see sample code}
// create String representation of card
 public String toString()
           {return getRank() + " of " + getSuit();}
}
```

Improvements to Card52

Add back a constructor with Rank/Suit parameters

```
public class Card52v2 implements Card {
...
public Card52v2( Rank theRank, Suit theSuit) {
  code = theSuit.ordinal() * 13 + theRank.ordinal();
}
```

Replace switch statements in "get" methods...

```
public Suit getSuit() {
        return Suit.value( code / 13 );}
public Rank getRank() {
        return Rank.value( code % 13 );}
```

...by adding value() method to each enum

```
 public static Rank value(int ordVal) {
        return vals[ordVal];}
```

Interfaces: Worth Noting

- Interface methods **are always** public
  - Java does not allow non-public methods in interfaces
- Interface instance variables are always **static final**
  - static variables are shared across instances
  - final variables are constants: they can't change value
- Most classes contain constructors; interfaces do not!
- Can *declare* interface objects (just like class objects) but cannot instantiate ("new") them

Comparable Interface

- Java provides an interface for comparisons between objects
  - Provides a replacement for “<“ and “>” in recBinarySearch
- Java provides the *Comparable* interface, which specifies a method *compareTo()*
  - Any class that implements Comparable must provide compareTo()

```
public interface Comparable<T> {
    //post: return < 0 if this smaller than other
             return 0 if this equal to other
             return > 0 if this greater than other
      int compareTo(T other);
}
```

Comparable Interface

- Many Java-provided classes implement Comparable
  - String (alphabetical order)
  - Wrapper classes: Integer, Character, Boolean
  - All Enum classes
- We can write methods that work on any type that implements Comparable
  - Example: RecBinSearch.java and BinSearchComparable.java

compareTo in Card Example

We could write

```
public class CardRankSuit implements
      Comparable<CardRankSuit> {

   public int compareTo(CardRankSuit other) {
       if (this.getSuit() != other.getSuit())
          return getSuit().compareTo(other.Suit());
       else
          return getRank().compareTo(other.getRank());
   }
// rest of code for the class....
}
```

Comparable & compareTo

- The Comparable interface (Comparable<T>) is part of the java.lang (not structure5) package.
- Other Java-provided structures can take advantage of objects that implement Comparable
  - See the Arrays class in java.util
  - Example JavaArraysBinSearch
- Users of Comparable are urged to ensure that *compareTo()* and *equals()* are *consistent*. That is,
  - x.compareTo(y) == 0 exactly when x.equals(y) == true
- Note that Comparable limits user to a *single ordering*
- The syntax can get kind of dense
  - See BinSearchComparable.java : a generic binary search method
  - And even more cumbersome….

ComparableAssociation

- Suppose we want an *ordered* Dictionary, so that we can use binary search instead of linear
- Structure5 provides a ComparableAssociation class that implements Comparable.
- The class declaration for ComparableAssociation is

…wait for it...

public class ComparableAssociation<K extends Comparable<K>, V>
Extends Association<K,V> implements
Comparable<ComparableAssociation<K,V>>

(Yikes!)

- Example: Since Integer implements Comparable, we can write
  - ComparableAssociation<Integer, String> myAssoc =
    new ComparableAssociation(new Integer(567), “Bob”);
- We could then use Arrays.sort on an array of these

Comparators

- Limitations with Comparable interface
  - Only permits one order between objects
  - What if it isn't the desired ordering?
  - What if it isn't implemented?
- Solution: Comparators

Comparators (Ch 6.8)

- A comparator is an object that contains a method that is capable of comparing two objects
- Sorting methods can be written to apply a comparator to two objects when a comparison is to be performed
- Different comparators can be applied to the same data to sort in different orders or on different keys

```
public interface Comparator <E> {
     // pre: a and b are valid objects
     // post: returns a value <, =, or > than 0 determined by
     // whether a is less than, equal to, or greater than b
     public int compare(E a, E b);
}
```

Example

```
class Patient {
    protected int age;
    protected String name;
    public Patient (String s, int a) {name = s; age = a;}
    public String getName() { return name; }
    public int getAge() {return age;}
}
```

Note that Patient does not implement Comparable or Comparator!

```
class NameComparator implements Comparator <Patient>{
    public int compare(Patient a, Patient b) {
       return a.getName().compareTo(b.getName());
    }
} // Note: No constructor; a "do-nothing" constructor is added by Java
```

```
public void sort(T a[], Comparator<T> c) {
   …
   if (c.compare(a[i], a[max]) > 0) {…}
}
```

```
sort(patients, new NameComparator());
```

Comparable vs Comparator

- Comparable Interface for class X
  - Permits just one order between objects of class X
  - Class X must implement a compareTo method
  - Changing order requires rewriting compareTo
    - And recompiling class X
- Comparator Interface
  - Allows creation of "Compator classes" for class X
  - Class X isn't changed or recompiled
  - Multiple Comparators for X can be developed
    - Sort Strings by length (alphabetically for equal-length)

Selection Sort with Comparator

```
public static <E> int findPosOfMax(E[] a, int last,
              Comparator<E> c) {
       int maxPos = 0       // A wild guess
       for(int i = 1; i <= last; i++)
              if (c.compare(a[maxPos], a[i]) < 0) maxPos = i;
       return maxPos;
}
public static <E> void selectionSort(E[] a, Comparator<E> c) {
       for(int i = a.length - 1; i>0; i--) {
           int big= findPosOfMin(a,i,c);
           swap(a, i, big);
       }
}
```

- The same array can be sorted in multiple ways by passing different Comparator<E> values to the sort method;

Merge Sort

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

Merge Sort

- [8 14 29 1 17 39 16 9]
- [8 14 29 1] [17 39 16 9] split
- [8 14] [29 1] [17 39] [16 9] split
- [8] [14] [29] [1] [17] [39] [16] [9] split
- [8 14] [1 29] [17 39] [9 16] merge
- [1 8 14 29] [9 16 17 39] merge
- [1 8 9 14 16 17 29 39] merge

Merge Sort

- How would we implement 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

- How would we implement it?
  - Review MergeSort.java
  - 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 = O(n log n)

- [8 14 29 1 17 39 16 9]
- [8 14 29 1] [17 39 16 9] split
- [8 14] [29 1] [17 39] [16 9] split
- [8] [14] [29] [1] [17] [39] [16] [9] split
- [8 14] [1 29] [17 39] [9 16] merge
- [1 8 14 29] [9 16 17 39] merge
- [1 8 9 14 16 17 29 39] merge

log n (split)

log n (merge)

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 case $k \leq 1$: 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
- $T(k) \leq 2^k + 2 * T(k-1)$

$$\leq 2^k + 2(k-1)2^{k-1} \leq k2^k$$

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 problems or limitations with Merge sort?
- Why would we ever use any other algorithm for sorting?

Problems with 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

Recall Merge Sort

```
private static void mergeSortRecursive(Comparable data[],
                        Comparable temp[], int low, int high) {
   int n = high-low+1;
   int middle = low + n/2;
   int i;

   if (n < 2) return;
   // move lower half of data into temporary storage
   for (i = low; i < middle; i++) {
       temp[i] = data[i];
   }
   // sort lower half of array
   mergeSortRecursive(temp,data,low,middle-1);
   // sort upper half of array
   mergeSortRecursive(data,temp,middle,high);
   // merge halves together
   merge(data,temp,low,middle,high);
}
```

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

1. Put first element (pivot) into sorted position
2. All to the left of “pivot” are smaller and all to the right are larger
3. Return index of “pivot”

Partition

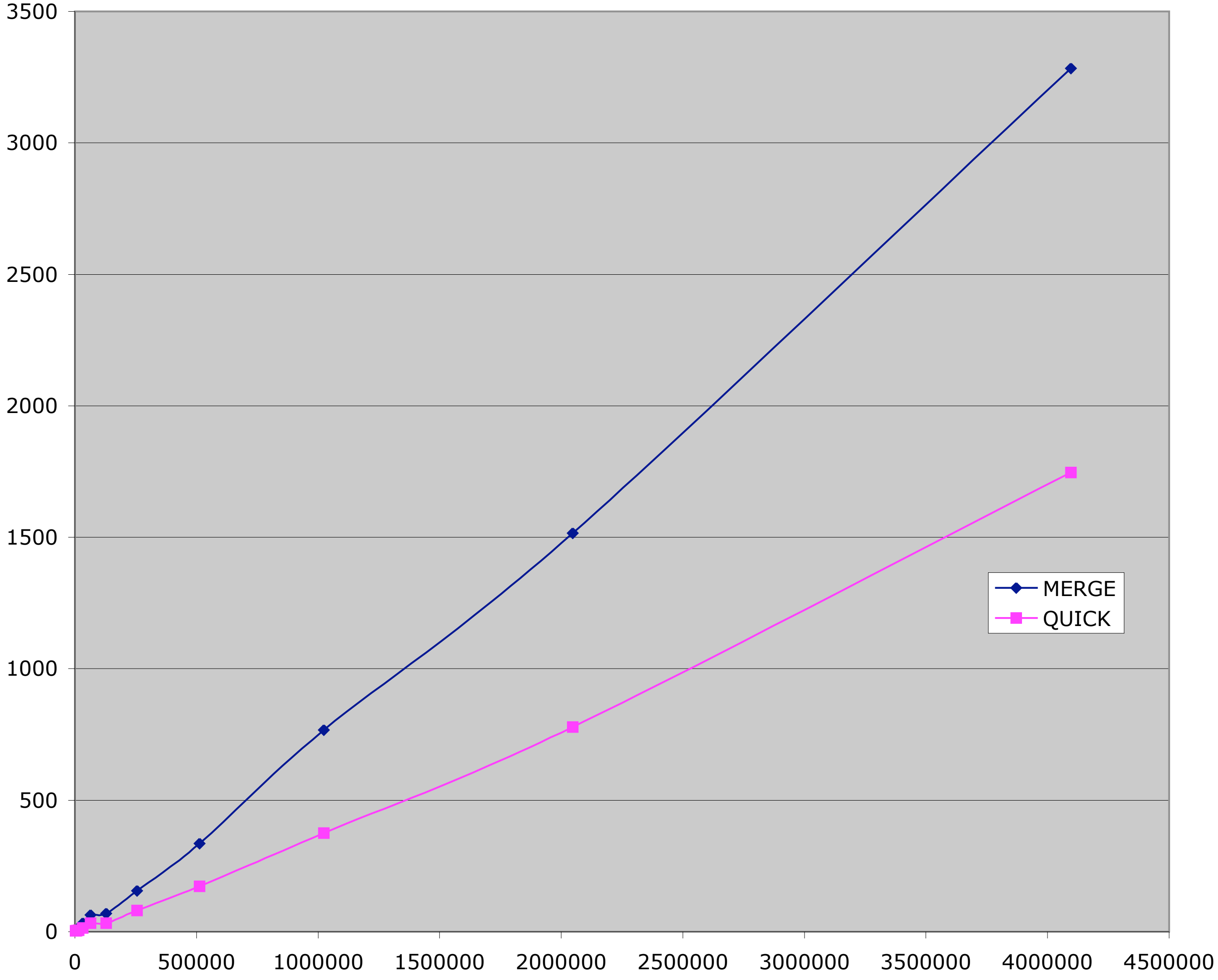
```
int partition(int data[], int left, int right) {
  while (true) {
    while (left < right && data[left] < data[right])
      right--;
    if (left < right) {
      swap(data,left++,right);
    } else {
      return left;
    }

    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



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: $O(n^2)$	$O(n) : n + c$

More Skill-Testing (Try these at home)

Given the following list of integers:

9 5 6 1 10 15 2 4

1) Sort the list using Insertion sort. . Show your work!
2) Sort the list using Merge sort. . Show your work!
3) Verify the best and worst case time and space complexity for each of these sorting algorithms as well as for selection sort.