"Organizing is what you do before you do something, so that when you do it, it is not all mixed up.”

~ A. A. Milne

**Prelim 1**
- It’s on Tuesday Evening (3/13)
- Two Sessions:
  - 5:30-7:00PM: netid aa..ks
  - 7:30-9:00PM: netid kt..zz
- If you have a conflict with your assigned time but can make the other time, fill out conflict assignment on CMS BY TOMORROW
- Three Rooms:
  - We will email you Tuesday morning with your room
  - Bring your Cornell ID!!!

**Why Sorting?**
- Sorting is useful
  - Database indexing
  - Operations research
  - Compression
- There are lots of ways to sort
  - There isn’t one right answer
  - You need to be able to figure out the options and decide which one is right for your application.
  - Today, we’ll learn about several different algorithms (and how to derive them)

**Some Sorting Algorithms**
- Insertion sort
- Selection sort
- Merge sort
- Quick sort

**InsertionSort**
```
pre: b[0..i-1] is sorted
post: b[0..i] sorted
```

A loop that processes elements of an array in increasing order has this invariant.
Each iteration, i = i+1; How to keep inv true?

<table>
<thead>
<tr>
<th>inv:</th>
<th>b[0]</th>
<th>i</th>
<th>b.length</th>
</tr>
</thead>
<tbody>
<tr>
<td>e.g.</td>
<td>b[0]</td>
<td>i</td>
<td>b.length</td>
</tr>
<tr>
<td></td>
<td>2 5 5 5 7</td>
<td>3</td>
<td>?</td>
</tr>
</tbody>
</table>

Loop body (inv true before and after)

<table>
<thead>
<tr>
<th>inv:</th>
<th>b[0]</th>
<th>i</th>
<th>b.length</th>
</tr>
</thead>
<tbody>
<tr>
<td>e.g.</td>
<td>b[0]</td>
<td>i</td>
<td>b.length</td>
</tr>
<tr>
<td></td>
<td>2 5 5 3 5</td>
<td>7</td>
<td>?</td>
</tr>
</tbody>
</table>

This will take time proportional to the number of swaps needed

What to do in each iteration?

<table>
<thead>
<tr>
<th>inv:</th>
<th>b[0]</th>
<th>i</th>
<th>b.length</th>
</tr>
</thead>
<tbody>
<tr>
<td>e.g.</td>
<td>b[0]</td>
<td>i</td>
<td>b.length</td>
</tr>
<tr>
<td></td>
<td>2 5 5 7</td>
<td>3</td>
<td>?</td>
</tr>
</tbody>
</table>

Push b[i] to its sorted position in b[0..i], then increase i

What to do in each iteration?

<table>
<thead>
<tr>
<th>inv:</th>
<th>b[0]</th>
<th>i</th>
<th>b.length</th>
</tr>
</thead>
<tbody>
<tr>
<td>e.g.</td>
<td>b[0]</td>
<td>i</td>
<td>b.length</td>
</tr>
<tr>
<td></td>
<td>2 5 5 3 5</td>
<td>7</td>
<td>?</td>
</tr>
</tbody>
</table>

Push b[i] to its sorted position in b[0..i], then increase i

This will take time proportional to the number of swaps needed

Insertion Sort

// sort b[], an array of int
// inv: b[0..i-1] is sorted
for (int i = 0; i < b.length; i++) {
    // Push b[i] down to its sorted position in b[0..i]
    // Present algorithm like this
    
    This is the best way to present it. We expect you to present it this way when asked.

    Later, can show how to implement that with an inner loop

Insertion Sort

// sort b[], an array of int
// inv: b[0..i-1] is sorted
for (int i = 0; i < b.length; i++) {
    // Push b[i] down to its sorted position in b[0..i]
    int k = i;
    while (k > 0 && b[k] < b[k-1]) {
        Swap b[k] and b[k-1]
        k = k-1;
    }
}

Let n = b.length

\[ \text{Pushing } b[i] \text{ down can take } i \text{ swaps.} \]

Worst case takes

\[ 1 + 2 + 3 + \ldots + (n-1) = \frac{(n-1)n}{2} \]

Expected case: \( O(n^2) \)

Performance

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Time</th>
<th>Space</th>
<th>Stable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insertion Sort</td>
<td>( O(n^2) )</td>
<td>( O(n^2) )</td>
<td>Yes</td>
</tr>
<tr>
<td>Selection Sort</td>
<td>( O(n^2) )</td>
<td>( O(1) )</td>
<td>No</td>
</tr>
<tr>
<td>Merge Sort</td>
<td>( O(n) )</td>
<td>( O(1) )</td>
<td>Yes</td>
</tr>
<tr>
<td>Quick Sort</td>
<td>( O(n \log n) )</td>
<td>( O(1) )</td>
<td>No</td>
</tr>
</tbody>
</table>
Selection Sort

```
// Selection Sort

// pre: b[0..b.length-1] unsorted
// post: b[0..b.length-1] sorted

// invariant:
// c[0..i-1] sorted, b[i..b.length-1] <= b[i]
// c[0..i-1] <= b[i..b.length-1]
// c[0..i-1] <= c[0..i-1]

for (int i = 0; i < b.length; i++) {
  int m = index of minimum of b[i..b.length-1];
  Swap b[i] and b[m];
}
```

**Performance**

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Time</th>
<th>Space</th>
<th>Stable?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insertion Sort</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>Yes</td>
</tr>
<tr>
<td>Selection Sort</td>
<td>$O(n^2)$</td>
<td>$O(1)$</td>
<td>No</td>
</tr>
<tr>
<td>Merge Sort</td>
<td>$O(n \log n)$</td>
<td>$O(n)$</td>
<td>Yes</td>
</tr>
<tr>
<td>Quick Sort</td>
<td>$O(n \log n)$</td>
<td>$O(n)$</td>
<td>No</td>
</tr>
</tbody>
</table>

**Merge two adjacent sorted segments**

```
// Merge two adjacent sorted segments
/
/* Sort b[h..k]. Precondition: b[h..t] and b[t+1..k] are sorted. */
/ public static merge(int[] b, int h, int t, int k) {
  Copy b[h..t] into a new array c;
  Merge c and b[t+1..k] into b[h..k];
} `
Merge

```java
int i = 0;
int u = h;
int v = t+1;
while (i <= t - h){
    if(v < k && b[v] < c[i]) {
        b[u] = b[v];
        u++; v++;
    }else {
        b[u] = c[i];
        u++; i++;
    }
}
```

Mergesort

```java
/** Sort b[h..k] */
public static void mergesort(int[] b, int h, int k) {
    if (size b[h..k] < 2)
        return;
    int t= (h+k)/2;
    mergesort(b, h, t);
    mergesort(b, t+1, k);
    merge(b, h, t, k);
}
```

Performance

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Time</th>
<th>Space</th>
<th>Stable?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insertion Sort</td>
<td>(O(n^2))</td>
<td>(O(1))</td>
<td>Yes</td>
</tr>
<tr>
<td>Selection Sort</td>
<td>(O(n^2))</td>
<td>(O(1))</td>
<td>No</td>
</tr>
<tr>
<td>Merge Sort</td>
<td>(n \log(n))</td>
<td>(O(n))</td>
<td>Yes</td>
</tr>
<tr>
<td>Quick Sort</td>
<td>(n \log(n))</td>
<td>(O(n))</td>
<td>Yes</td>
</tr>
</tbody>
</table>

QuickSort

QuickSort developed by Sir Tony Hoare (he was knighted by the Queen of England for his contributions to education and CS).
83 years old.
Developed QuickSort in 1958. But he could not explain it to his colleague, so he gave up on it.
Later, he saw a draft of the new language Algol 58 (which became Algol 60). It had recursive procedures. First time in a procedural programming language. “Ah!” he said. “I know how to write it better now.” 15 minutes later, his colleague also understood it.

Partition algorithm of quicksort

pre: h h+1 ? k
post: <= x x >= x

x is called the pivot

Swap array values around until b[h..k] looks like this:

h j k

Not yet sorted
these can be in any order
Not yet sorted
these can be in any order
The 20 could be in the other partition
### Partition Algorithm

**Pre:**

- \( h \) \( h+1 \) \( k \)
- Combine pre and post to get an invariant

**Post:**

- \( b \) \( h \) \( j \) \( t \) \( k \)

**Invariant:**

- \( h \) \( j \) \( t \) \( k \)

Initially, with \( j = h \) and \( t = k \), this diagram looks like the start diagram.

**Terminate when \( j = t \), so the "?" segment is empty, so diagram looks like result diagram.**

### QuickSort Procedure

```java
/** Sort b[h..k]. */
public static void QS(int[] b, int h, int k) {
    if (b[h..k] has < 2 elements) return; // Base case
    int j = partition(b, h, k);
    QS(b, h, j-1);
    QS(b, j+1, k);
}
```

- **Function does the partition algorithm and returns position \( j \) of pivot**
- **Max depth: \( O(\log n) \). Time to partition on each level: \( O(n) \)**
- **Total time: \( O(n \log n) \).**
- **Average time for Quicksort: \( n \log n \). Difficult calculation**

### Worst case quicksort: pivot always smallest value

- **Worst-case: \( O(n^2) \) space: \( ? \)**
- **What's depth of recursion?**

### QuickSort Complexity to sort array of length \( n \)

```java
/** Sort b[h..k]. */
public static void QS(int[] b, int h, int k) {
    if (b[h..k] has < 2 elements) return;
    int j = partition(b, h, k);
    QS(b, h, j-1);
    QS(b, j+1, k);
}
```

- **Worst-case space: \( O(n) \)**
- **What's depth of recursion?**

- **Time complexity**
  - **Worst-case: \( O(n^2) \)**
  - **Average-case: \( O(n \log n) \)**
QuickSort versus MergeSort

/** Sort b[h..k] */
public static void QS (int[] b, int h, int k) {
  if (k - h < 1) return;
  int j = partition(b, h, k);
  QS(b, h, j - 1);
  QS(b, j + 1, k);
}

/** Sort b[h..k] */
public static void MS (int[] b, int h, int k) {
  if (k - h < 1) return;
  MS(b, h, (h + k) / 2);
  MS(b, (h + k) / 2 + 1, k);
  merge(b, h, (h + k) / 2, k);
}

One processes the array then recurses.
One recurses then processes the array.

Partition. Key issue. How to choose pivot

Choosing pivot
Ideal pivot: the median, since it splits array in half
But computing is $O(n)$, quite complicated

Popular heuristics: Use
- first array value (not so good)
- middle array value (not so good)
- Choose a random element (not so good)
- median of first, middle, last, values (often used)

Performance

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Time</th>
<th>Space</th>
<th>Stable?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insertion Sort</td>
<td>$O(n)$ to $O(n^2)$</td>
<td>$O(1)$</td>
<td>Yes</td>
</tr>
<tr>
<td>Selection Sort</td>
<td>$O(n^2)$</td>
<td>$O(1)$</td>
<td>No</td>
</tr>
<tr>
<td>Merge Sort</td>
<td>$n \log(n)$</td>
<td>$O(n)$</td>
<td></td>
</tr>
<tr>
<td>Quick Sort</td>
<td>$n \log(n)$ to $O(n^2)$</td>
<td>$O(\log(n))$</td>
<td>No</td>
</tr>
</tbody>
</table>

Sorting in Java

- Java.util.Arrays has a method Sort()
  - implemented as a collection of overloaded methods
  - for primitives, Sort is implemented with a version of quicksort
  - for Objects that implement Comparable, Sort is implemented with mergesort
- Tradeoff between speed/space and stability/performance guarantees

QuickSort with logarithmic space

Problem is that if the pivot value is always the smallest (or always the largest), the depth of recursion is the size of the array to sort.

Eliminate this problem by doing some of it iteratively and some recursively. We may show you this later. Not today!

QuickSort with logarithmic space

/** Sort b[h..k]. */
public static void QS(int[] b, int h, int k) {
  int h1 = h; int k1 = k;
  // invariant b[h..k] is sorted if b[h1..k1] is sorted
  while (b[h1..k1] has more than 1 element) {
    Reduce the size of b[h1..k1], keeping inv true
  }
}
QuickSort with logarithmic space

```java
/** Sort b[h..k]. */
public static void QS(int[] b, int h, int k) {
    int h1 = h; int k1 = k;
    // invariant b[h..k] is sorted if b[h1..k1] is sorted
    while (b[h1..k1] has more than 1 element) {
        int j = partition(b, h1, k1);
        // b[h1..j-1] <= b[j] <= b[j+1..k1]
        if (b[h1..j-1] smaller than b[j+1..k1])
            QS(b, h, j-1); h1 =  j+1;
        else
            QS(b, j+1, k1); k1 =  j-1;
    }
}
```

Only the smaller segment is sorted recursively. If b[h1..k1] has size n, the smaller segment has size < n/2. Therefore, depth of recursion is at most log n.