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

~ A. A. Milne

Announcements

Apply to be a teacher!

is a program with a “teach anything, learn anything” philosophy. You will be able to provide high schoolers with instruction in the topic of your choice.

This semester’s event is on Saturday, November 4

Apply to be a teacher!

If you are interested, please email us at: splashcornell@gmail.com.

Prelim 1

- It’s on Thursday Evening (9/28)
- Two Sessions:
  - 5:30-7:00PM: A..Lid
  - 7:30-9:00PM: Lie..Z
- Three Rooms:
  - We will email you Thursday morning with your room
- Bring your Cornell ID!!!

A3

A3 Comments

- good
- linked
- understand
- learn
- think
- really
- interesting
- data
- difficult
- testing
- doubly
- list
- method
- first
- one
- code
- formatting
- made
A3 Comments

/* Mini lecture on linked lists would have been very helpful. I still do not
know when we covered this topic in class. It was initially difficult to
understand what we were meant to do without having learned the topic
in depth before */

/* Maybe the assignment guide could explain a bit more about how to
thoroughly test the methods through. Testing is still a bit difficult and I
wish we had an assignment which covered that more. The instructions
could have been more specific about what is expected from the test
cases though. */

/* It also showed me how important it is to test after writing a method. I
had messed up on one of the earlier methods and if I had waited to test
I would have had a lot of trouble figuring out what went wrong. This
assignment showed me how vital it is to test not at the end but
incrementally. I feel more careful, efficient, and organized. */

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

<table>
<thead>
<tr>
<th>pre: b</th>
<th>post: b</th>
<th>sorted</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>b.length</td>
<td>?</td>
</tr>
</tbody>
</table>

```
inv: b | i | b.length |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>sorted</td>
<td>?</td>
<td></td>
</tr>
</tbody>
</table>
```

```
or: b[0..i-1] is sorted
```

```
<table>
<thead>
<tr>
<th>b</th>
<th>i</th>
<th>b.length</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 5 5 5 7</td>
<td>3</td>
<td>?</td>
</tr>
</tbody>
</table>
```

```
e.g. b | i | b.length |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2 3 5 5 5 7</td>
<td>?</td>
<td></td>
</tr>
</tbody>
</table>
```

What to do in each iteration?

```
inv: b | i | b.length |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>sorted</td>
<td>?</td>
<td></td>
</tr>
</tbody>
</table>
```

```
e.g. b | i | b.length |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2 5 5 5 7 3</td>
<td>?</td>
<td></td>
</tr>
</tbody>
</table>
```

```
Loop body
(inv true before and after)
```

```
<table>
<thead>
<tr>
<th>b</th>
<th>i</th>
<th>b.length</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 5 5 5 3</td>
<td>?</td>
<td></td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>b</th>
<th>i</th>
<th>b.length</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 5 3 5 5 7</td>
<td>?</td>
<td></td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>b</th>
<th>i</th>
<th>b.length</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 3 5 5 5 7</td>
<td>?</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

```
b | i | b.length |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2 3 5 5 5</td>
<td>?</td>
<td></td>
</tr>
</tbody>
</table>
```
Insertion Sort

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

Note English statement in body. Abstraction. Says what to do, not how. 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.

```
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)</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></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Quick Sort</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

SelectionSort

```
// sort b[], an array of int
// inv: b[0..i-1] sorted AND b[0..i-1] <= b[i..]
for (int i = 0; i < b.length; i = i+1) {
    int m= index of minimum of b[i..];
    Swap b[i] and b[m];
}
```

Additional term in invariant

```
Keep invariant true while making progress?
```

```
Increasing i by 1 keeps inv true only if b[i] is min of b[..i]
```

Another common way for people to sort cards

```
Runtime
with n = b.length
- Worst-case O(n^2)
- Best-case O(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)) (\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></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Quick Sort</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Insertion Sort**
  - Time: \(O(n)\) \(\to \ O(n^2)\)
  - Space: \(O(1)\)
  - Stable: Yes

- **Selection Sort**
  - Time: \(O(n^2)\)
  - Space: \(O(1)\)
  - Stable: No

---

**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) {
  
  
  }

- 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) {
  
  
  }

- Merge two adjacent sorted segments

- // Merge sorted c and b[t+1..k] into b[h..k]
- pre: c = x, b = y, x, y are sorted
- post: b[h..k] is sorted
- invariant: c[0..c.length] = x and y, sorted
- c.length = i = 0
- x = h
- y = t
- k = k

- **Mergesort**

- /** Sort b[h..k] */
- public static void mergesort(int[] b, int h, int k) {
  
  
  }

- Mergesort

- /** Sort b[h..k] */
- public static void mergesort(int[] b, int h, int k) {
  
  
  }

- Mergesort

- /* 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) {
  
  
  }

- 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) {
  
  
  }

- Merge two adjacent sorted segments

- // Merge sorted c and b[t+1..k] into b[h..k]
- pre: c = x, b = y, x, y are sorted
- post: b[h..k] is sorted
- invariant: c[0..c.length] = x and y, sorted
- c.length = i = 0
- x = h
- y = t
- k = k

- **Mergesort**

- /** Sort b[h..k] */
- public static void mergesort(int[] b, int h, int k) {
  
  
  }

- Mergesort

- /** Sort b[h..k] */
- public static void mergesort(int[] b, int h, int k) {
  
  
  }

- Mergesort

- /* 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) {
  
  
  }

- 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) {
  
  
  }

- Merge two adjacent sorted segments

- // Merge sorted c and b[t+1..k] into b[h..k]
- pre: c = x, b = y, x, y are sorted
- post: b[h..k] is sorted
- invariant: c[0..c.length] = x and y, sorted
- c.length = i = 0
- x = h
- y = t
- k = k

- **Mergesort**

- /** Sort b[h..k] */
- public static void mergesort(int[] b, int h, int k) {
  
  
  }

- Mergesort

- /** Sort b[h..k] */
- public static void mergesort(int[] b, int h, int k) {
  
  
  }

- Mergesort

- /* 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) {
  
  
  }

- 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) {
  
  
  }

- Merge two adjacent sorted segments

- // Merge sorted c and b[t+1..k] into b[h..k]
- pre: c = x, b = y, x, y are sorted
- post: b[h..k] is sorted
- invariant: c[0..c.length] = x and y, sorted
- c.length = i = 0
- x = h
- y = t
- k = k

- **Mergesort**

- /** Sort b[h..k] */
- public static void mergesort(int[] b, int h, int k) {
  
  
  }

- Mergesort

- /** Sort b[h..k] */
- public static void mergesort(int[] b, int h, int k) {
  
  
  }

- Mergesort

- /* 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) {
  
  
  }

- 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) {
  
  
  }

- Merge two adjacent sorted segments

- // Merge sorted c and b[t+1..k] into b[h..k]
- pre: c = x, b = y, x, y are sorted
- post: b[h..k] is sorted
- invariant: c[0..c.length] = x and y, sorted
- c.length = i = 0
- x = h
- y = t
- k = k

- **Mergesort**

- /** Sort b[h..k] */
- public static void mergesort(int[] b, int h, int k) {
  
  
  }

- Mergesort

- /** Sort b[h..k] */
- public static void mergesort(int[] b, int h, int k) {
  
  
  }

- Mergesort

- /* 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) {
  
  
  }

- 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) {
  
  
  }

- Merge two adjacent sorted segments

- // Merge sorted c and b[t+1..k] into b[h..k]
- pre: c = x, b = y, x, y are sorted
- post: b[h..k] is sorted
- invariant: c[0..c.length] = x and y, sorted
- c.length = i = 0
- x = h
- y = t
- k = k

- **Mergesort**

- /** Sort b[h..k] */
- public static void mergesort(int[] b, int h, int k) {
  
  
  }

- Mergesort

- /** Sort b[h..k] */
- public static void mergesort(int[] b, int h, int k) {
  
  
  }

- Mergesort

- /* 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) {
  
  
  }

- 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) {
  
  
  }

- Merge two adjacent sorted segments

- // Merge sorted c and b[t+1..k] into b[h..k]
- pre: c = x, b = y, x, y are sorted
- post: b[h..k] is sorted
- invariant: c[0..c.length] = x and y, sorted
- c.length = i = 0
- x = h
### 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>( \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:**

\[ x \]

\[ h \]

\[ h+1 \]

\[ k \]

\[ ? \]

\[ x \]

\[ \]

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

**Post:**

\[ h \]

\[ j \]

\[ k \]

\[ \]

\[ <= x \]

\[ x \]

\[ >= x \]

- \( x \) is called the pivot

### Partition algorithm

**Pre:**

\[ h \]

\[ j \]

\[ t \]

\[ k \]

\[ \]

\[ <= x \]

\[ x \]

\[ ? \]

\[ >= x \]

- Combine pre and post to get an invariant

**Post:**

\[ h \]

\[ j \]

\[ t \]

\[ k \]

\[ \]

\[ <= x \]

\[ x \]

\[ ? \]

\[ >= x \]

- 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);
    // We know b[h..j-1] <= b[j] <= b[j+1..k]
    // Sort b[h..j-1] and b[j+1..k]
    QS(b, h, j-1);
    QS(b, j+1, k);
}
```

Best case quicksort: pivot always middle value

```java
0
```

QuickSort versus MergeSort

```java
/** 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);
    // Sort b[h..j-1] and b[j+1..k]
    QS(b, h, j-1);
    QS(b, j+1, k);
}
```

Worst case quicksort: pivot always smallest value

```java
j
```

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);
    // We know b[h..j-1] <= b[j] <= b[j+1..k]
    // Sort b[h..j-1] and b[j+1..k]
    QS(b, h, j-1);
    QS(b, j+1, k);
}
```

Partition. Key issue. How to choose pivot

```java
pre: h h k
    Choosing pivot
    Ideal pivot: the median, since it splits array in half
    But computing is O(n), quite complicated
post: b <= x x >= x
```

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>Yes</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 merge sort
- 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

```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 \(\log n\).