"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

SPASH! at Cornell 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.
Announcements

**SLOPE DAY**
**G-BODY MEETING**
**9/26 @ 4:30P**
**URIS 262**
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 Comments

linked lists, methods, data structures, fun, practice, testing, first, overall, helpful, understanding, thinking, easy, interesting, data, difficult, writing, difficulty, definitely, make, took, know, work, liked, simple, functions, something, confusing, made, overall, introduction, help, important, introduced, part, before, write, now, doubly, know, made, code, practice, nodes, fun, structure, using, A3, however, implement, just, methods, helpful, draw, still, wish, helps, think, good, understand, straightforward, syntax, got, everything, actual, cases, also, sure, get, node, classes, feel, clear, learning, way, great, especially, function, felt, thinking, little, helped, much, experience, inner, like, Pretty, hard, through, use, previous, instructions, though, use, assignment, to String, Rev, different, DLL, concepts, worked, able, node, test, lot, like, feel, get, classes, understand, understand, list, method, useful, Linked, write, difficulty, never, already, start, assert, quite, Node, coding, append, however, difficult, diagrams, bit, challenging, using, A3, now, doubly, know, work, liked, one, simple, functions, something, confusing, made, overall.
/* 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 though. 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**

A loop that processes elements of an array in increasing order has this invariant:

- **pre:** \( b[0..i-1] \) is sorted
- **inv:** \( b[0..i] \) is sorted, \( b[i] \) is the next element to be inserted
- **post:** \( b[0..i] \) is sorted
Each iteration, \( i = i + 1 \); How to keep \( \text{inv} \) true?

<table>
<thead>
<tr>
<th>( i )</th>
<th>( \text{sorted} )</th>
<th>( \text{?} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2 5 5 5 7</td>
<td>3</td>
</tr>
<tr>
<td>b</td>
<td>?</td>
<td>?</td>
</tr>
</tbody>
</table>

**inv:**

1. Initialize \( i = 0 \), \( \text{sorted} \), and \( \text{?} \).
2. For each iteration, increment \( i \).
3. Ensure \( \text{sorted} \) is maintained.
4. Update \( \text{?} \) accordingly.

**e.g.:**

1. Start with \( i = 0 \), \( \text{sorted} = \) true, \( \text{?} = \) true.
2. Increment \( i \) and ensure \( \text{sorted} \) is still true.
3. Update \( \text{?} \) if necessary.

**b.length:**

- Represents the length of the array.
- Used to check bounds and conditions.
What to do in each iteration?

<table>
<thead>
<tr>
<th>0</th>
<th>i</th>
<th>b.length</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>sorted</td>
<td>?</td>
</tr>
</tbody>
</table>

**inv:**

<table>
<thead>
<tr>
<th>2</th>
<th>5</th>
<th>5</th>
<th>5</th>
<th>7</th>
<th>3</th>
<th>?</th>
</tr>
</thead>
</table>

**e.g.:**

| 2 | 5 | 5 | 5 | 7 | 3 | ? |

**Loop body (inv true before and after):**

- 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

| 2 | 3 | 5 | 5 | 5 | 7 | ? |
// 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]

}
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]
    int k = i;
    while (k > 0 && b[k] < b[k-1]) {
        Swap b[k] and b[k-1]
        k= k−1;
    }
}

invariant P: b[0..i] is sorted
except that b[k] may be < b[k-1]

\[\begin{array}{ccccccc}
2 & 5 & 3 & 5 & 5 & 7 & ?
\end{array}\]

example

start?
stop?
progress?
maintain
invariant?
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]
}

Pushing b[i] down can take i swaps.
Worst case takes

1 + 2 + 3 + ... n-1 = (n-1)*n/2

Swaps.

Let n = b.length

• Worst-case: O(n^2)
  (reverse-sorted input)

• Best-case: O(n)
  (sorted input)

• 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>
SelectionSort

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

<table>
<thead>
<tr>
<th>inv:</th>
</tr>
</thead>
<tbody>
<tr>
<td>b[0..b.length] sorted, (\leq b[i..]) (\geq b[0..i-1])</td>
</tr>
</tbody>
</table>

Keep invariant true while making progress?

<table>
<thead>
<tr>
<th>e.g.:</th>
</tr>
</thead>
<tbody>
<tr>
<td>b[0..b.length]</td>
</tr>
</tbody>
</table>

| Increasing i by 1 keeps inv true only if b[i] is min of b[i..] |
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];
}
## 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>
/* 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) {
    Copy b[h..t] into a new array c;
    Merge c and b[t+1..k] into b[h..k];
}

h  t  k
b  4 7 7 8 9 3 4 7 8

h  sorted  t  sorted
k

b  3 4 4 7 7 7 8 8 9

h  merged, sorted
k
Merge two adjacent sorted segments

// Merge sorted c and b[t+1..k] into b[h..k]

pre:  

\[
\begin{array}{cccc}
0 & t-h & h & t & k \\
c & x & b & ? & y \\
\end{array}
\]

x, y are sorted

post: b

\[
\begin{array}{cccc}
h & h & k \\
\end{array}
\]

x and y, sorted

invariant:  

\[
\begin{array}{cccc}
0 & i & c.length \\
c & head of x & tail of x \\
\end{array}
\]

\[
\begin{array}{cccc}
h & u & v & k \\
b & ? & tail of y \\
\end{array}
\]

head of x and head of y, sorted

x, y are sorted
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++;
    }
}

pre: c
    sorted
    b
    ?
    sorted

post: b
    sorted

inv: 0
    i
    c.length

b
    ?
    v
    k

sorted

sorted

sorted

sorted
/** 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>$O(n)$ 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></td>
<td></td>
<td></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: 

\[
\begin{array}{c|c|c|c}
\text{h} & \text{h+1} & \text{k} \\
\text{x} & ? & \text{x is called the pivot} \\
\end{array}
\]

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

post: 

\[
\begin{array}{c|c|c|c}
\text{h} & \text{j} & \text{k} \\
\leq \text{x} & \text{x} & \geq \text{x} \\
\end{array}
\]
Not yet sorted

these can be in any order

The 20 could be in the other partition
### Partition algorithm

<table>
<thead>
<tr>
<th>pre:</th>
<th>post:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>b</strong></td>
<td><strong>b</strong></td>
</tr>
<tr>
<td>h</td>
<td>h</td>
</tr>
<tr>
<td>h+1</td>
<td>j</td>
</tr>
<tr>
<td>k</td>
<td>k</td>
</tr>
<tr>
<td>x</td>
<td>&lt;= x</td>
</tr>
<tr>
<td>?</td>
<td>x</td>
</tr>
<tr>
<td>&gt;= x</td>
<td>&gt;= x</td>
</tr>
</tbody>
</table>

Combine pre and post to get an invariant

<table>
<thead>
<tr>
<th>b</th>
<th>&lt;= x</th>
<th>x</th>
<th>?</th>
<th>&gt;= x</th>
</tr>
</thead>
<tbody>
<tr>
<td>h</td>
<td>j</td>
<td>t</td>
<td>k</td>
<td></td>
</tr>
</tbody>
</table>
Partition algorithm

j = h; t = k;
while (j < t) {
    if (b[j + 1] <= b[j]) {
        Swap b[j + 1] and b[j];  j = j + 1;
    } else {
        Swap b[j + 1] and b[t];  t = t - 1;
    }
}

Terminate when j = t, so the “?” segment is empty, so diagram looks like result diagram

Takes linear time: O(k+1-h)

Initially, with j = h and t = k, this diagram looks like the start diagram
QuickSort procedure

/** 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);
}

h j k
<= x x >= x
Worst case quicksort: pivot always smallest value

$$\begin{array}{c|c}
\text{x0} & \geq \text{x0} \\
\hline
\text{x0} & \text{x1} & \geq \text{x1} \\
\hline
\text{x0} & \text{x1} & \text{x2} & \geq \text{x2} \\
\end{array}$$

partitioning at depth 0
partitioning at depth 1
partitioning at depth 2

/** 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);
}

Depth of recursion: O(n)
Processing at depth i: O(n-i)
O(n*n)
Best case quicksort: pivot always middle value

Depth 0. 1 segment of size ~n to partition.

Depth 2. 2 segments of size ~n/2 to partition.

Depth 3. 4 segments of size ~n/4 to partition.

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
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);
}
```

Time complexity
Worst-case: $O(n^2)$
Average-case: $O(n \log n)$

Worst-case space: ??
What’s depth of recursion?

Worst-case space: $O(n)!$
--depth of recursion can be $n$
Can rewrite it to have space $O(\log n)$
Show this at end of lecture if we have time
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);
    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

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)

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

/** 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; }
    }
}