What to do in each iteration?

What to do in each iteration?

InsertionSort

- Worst-case: O(n²)  
- Best-case: O(n)  
- Expected case: O(n²)

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

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

Swaps.

Let n = b.length

SelectionSort

- Worst-case: O(n²)  
- Best-case: O(n)  
- Expected case: O(n²)

Keep invariant true while making progress?

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

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

This is the best way to present it. Later, show how to implement that with a loop.

Many people sort cards this way Works well when input is nearly sorted
SelectionSort

Another common way for people to sort cards

Runtime
- Worst-case $O(n^2)$
- Best-case $O(n^2)$
- Expected-case $O(n^2)$

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

b

sorted, smaller values
larger values

Each iteration, swap min value of this section into b[i]

Partition algorithm of quicksort

Idea
Using the pivot value $x$ that is in $b[h]$:

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

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

Not yet sorted

The 20 could be in the other partition

Partition algorithm

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

Not yet sorted

Not yet sorted

Partition algorithm

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

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

```
<= x  x  ?  >= x
```

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

```
<= x  x  ?  >= x
```

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]
}
```

Function does the partition algorithm and returns position $j$ of pivot
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;
    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);
}
```

Worst-case: quadratic
Average-case: O(n log n)
Worst-case space: O(n) -- depth of recursion can be n
Can rewrite it to have space O(log n)
Average-case: O(log n)

Best case quicksort: pivot always middle value

<table>
<thead>
<tr>
<th>j</th>
<th>j</th>
<th>j</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;= x0</td>
<td>x0</td>
<td>&gt;= x0</td>
</tr>
<tr>
<td>&lt;= x1</td>
<td>x1</td>
<td>&gt;= x2</td>
</tr>
<tr>
<td>&lt;= x2</td>
<td>x2</td>
<td>&gt;= x2</td>
</tr>
</tbody>
</table>

Max depth: about log n. Time to partition on each level: ~n
Total time: O(n log n).
Average time for Quicksort: n log n. Difficult calculation

Partition algorithm

Key issue: How to choose a pivot?
Choosing pivot
• Ideal pivot: the median, since it splits array in half
But computing median of unsorted array is O(n), quite complicated
Popular heuristics: Use
• first array value (not good)
• middle array value
• median of first, middle, last, values GOOD!
• Choose a random element

Worst case quicksort: pivot always smallest value

<table>
<thead>
<tr>
<th>j</th>
<th>j</th>
<th>j</th>
</tr>
</thead>
<tbody>
<tr>
<td>x0</td>
<td>x1</td>
<td>x2</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).
Will be 80 in April.
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 68 (which became Algol 60). It had recursive procedures. First time in a programming language. “Ah!,” he said. “I know how to write it better now.” 15 minutes later, his colleague also understood it.

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.
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.

```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.