Module 30

Searching & Sorting
def linear_search(v,b):
    
    """Returns: first occurrence of v in b (-1 if not found)
    Precond: b a list of number, v a number
    """
    # Loop variable
    i = 0
    while i < len(b) and b[i] != v:
        i = i + 1
    if i == len(b):  # not found
        return -1
    return i

How many entries do we have to look at?
def linear_search(v,b):

    """Returns: first occurrence of v in b (-1 if not found)
    Precond: b a list of number, v a number
    """

    # Loop variable
    i = 0
    while i < len(b) and b[i] != v:
        i = i + 1

    if i == len(b):  # not found
        return -1
    return i

How many entries do we have to look at?

All of them!
def linear_search(v,b):
    """Returns: first occurrence of v in b (-1 if not found)
    Precond: b a list of number, v a number
    """
    # Loop variable
    i = len(b)-1
    while i >= 0 and b[i] != v:
        i = i - 1
    # Equals -1 if not found
    return i

How many entries do we have to look at?
All of them!
Is There a Better Way?

• Thinking of number 0..100
  ▪ You get to guess number
  ▪ I tell you higher or lower
  ▪ Continue until get it right

• Goal: Keep # guesses low
  ▪ Use my answers to help

• Strategy?
  ▪ Start guess in the middle
  ▪ Answer eliminates half
  ▪ Go to middle of remaining
Is There a Better Way?

• Thinking of number 0..100
  ▪ You get to guess number
  ▪ I tell you higher or lower
  ▪ Continue until get it right

• Goal: Keep # guesses low
  ▪ Use my answers to help

• Strategy?
  ▪ Start guess in the middle
  ▪ Answer eliminates half
  ▪ Go to middle of remaining
Is There a Better Way?

• Thinking of number 0..100
  ▪ You get to guess number
  ▪ I tell you higher or lower
  ▪ Continue until get it right

• **Goal:** Keep # guesses low
  ▪ Use my answers to help

• **Strategy?**
  ▪ Start guess in the middle
  ▪ Answer eliminates half
  ▪ Go to middle of remaining
Is There a Better Way?

- Thinking of number 0..100
  - You get to guess number
  - I tell you higher or lower
  - Continue until get it right
- **Goal:** Keep # guesses low
  - Use my answers to help
- **Strategy?**
  - Start guess in the middle
  - Answer eliminates half
  - Go to middle of remaining

Higher!

0  50  62  75  100
Is There a Better Way?

- Thinking of number 0..100
  - You get to guess number
  - I tell you higher or lower
  - Continue until get it right

- **Goal:** Keep # guesses low
  - Use my answers to help

- **Strategy?**
  - Start guess in the middle
  - Answer eliminates half
  - Go to middle of remaining
def binary_search(v, b):
    # Loop variable(s)
    i = 0, j = len(b)
    while i < j and b[i] != v:
        mid = (i+j)//2
        if b[mid] < v:
            j = mid
        elif b[mid] > v:
            i = mid
        else:
            return mid
    return -1  # not found

Requires that the data is sorted!

But few checks!
Observation About Sorting

• Sorting data can speed up searching
  ▪ Sorting takes time, but do it once
  ▪ Afterwards, can search many times

• Not just searching. Also speeds up
  ▪ Duplicate elimination in data sets
  ▪ Data compression
  ▪ Physics computations in computer games

• Why it is a major area of computer science
The Sorting Challenge

- **Given:** A list of numbers
- **Goal:** Sort those numbers using only
  - Iteration (while-loops or for-loops)
  - Comparisons (< or >)
  - Assignment statements
- **Why?** For proper **analysis**.
  - Methods/functions come with hidden costs
  - Everything above has no hidden costs
  - Each comparison or assignment is “1 step”
This Requires Some Notation

• As the list is sorted…
  ▪ Part of the list will be sorted
  ▪ Part of the list will not be sorted

• Need a way to refer to portions of the list
  ▪ Notation to refer to sorted/unsorted parts

• And have to do it without slicing!
  ▪ Slicing makes a copy
  ▪ Want to sort original list, not a copy
This Requires Some Notation

- As the list is sorted...
  - Part of the list **will** be sorted
  - Part of the list will **not** be sorted
- Need a way to refer to portions of the list
  - Notation to refer to sorted/unsorted parts
- And have to do it **without** slicing!
  - Slicing makes a copy
  - Want to sort original list, not a copy

But we will be less formal than in past years!
Terminology: Range Notation

- $m..n$ is a range containing $n+1-m$ values
  - $2..5$ contains 2, 3, 4, 5. Contains $5+1-2 = 4$ values
  - $2..4$ contains 2, 3, 4. Contains $4+1-2 = 3$ values
  - $2..3$ contains 2, 3. Contains $3+1-2 = 2$ values
  - $2..2$ contains 2. Contains $2+1-2 = 1$ values
  - $2..1$ contains ???

What does $2..1$ contain?

A: nothing
B: 2,1
C: 1
D: 2
E: something else
Terminology: Range Notation

- \( m..n \) is a range containing \( n+1-m \) values
  - 2..5 contains 2, 3, 4, 5. Contains \( 5+1 - 2 = 4 \) values
  - 2..4 contains 2, 3, 4. Contains \( 4+1 - 2 = 3 \) values
  - 2..3 contains 2, 3. Contains \( 3+1 - 2 = 2 \) values
  - 2..2 contains 2. Contains \( 2+1 - 2 = 1 \) values
  - 2..1 contains ???

What does 2..1 contain?

- A: nothing
- B: 2,1
- C: 1
- D: 2
- E: something else
Terminology: Range Notation

- $m..n$ is a range containing $n+1-m$ values
  - 2..5 contains 2, 3, 4, 5. Contains $5+1-2=4$ values
  - 2..4 contains 2, 3, 4. Contains $4+1-2=3$ values
  - 2..3 contains 2, 3. Contains $3+1-2=2$ values
  - 2..2 contains 2. Contains $2+1-2=1$ values
  - 2..1 contains ???

- The notation $m..n$, always implies that $m \leq n+1$
  - So you can assume that even if we do not say it
  - If $m = n+1$, the range has 0 values
Horizontal Notation

- Want a pictoral way to visualize this sorting
  - Represent the list as long rectangle
  - We saw this idea in divide-and-conquer

- Do not show individual boxes
  - Just dividing lines between regions
  - Label dividing lines with indices
  - But index is either left or right of dividing line

\[(h+1) - h = 1\]
Horizontal Notation

• Label regions with properties
  - **Example**: Sorted or ???

```
0               k               n
b                sorted        ???
```

- b[0..k−1] is sorted
- b[k..n-1] **unknown** (might be sorted)

• Picture allows us to track progress
Visualizing Sorting

**Start:**

```
0
b
?  n
```

**Goal:**

```
0
b
sorted  n
```

**In-Progress:**

```
0
b
sorted  n
?  i
```
Insertion Sort

\[ i = 0 \]

While \( i < n \):

# Push \( b[i] \) down into its
# sorted position in \( b[0..i] \)

\[ i = i + 1 \]

Remember the restrictions!
**Insertion Sort: Moving into Position**

```python
i = 0
while i < n:
    push_down(b, i)
    i = i + 1

def push_down(b, i):
    j = i
    while j > 0:
        if b[j-1] > b[j]:
            swap(b, j-1, j)
            j = j - 1
```

Swap shown in the lecture about lists.
Insertion Sort: Moving into Position

\[ i = 0 \]

\[ \text{while } i < n: \]
\[ \quad \text{push\_down}(b, i) \]
\[ \quad i = i + 1 \]

```python
def push_down(b, i):
    j = i
    while j > 0:
        if b[j-1] > b[j]:
            swap(b, j-1, j)
        j = j - 1
```

\[
\begin{array}{ccccccc}
0 & 2 & 4 & 4 & 6 & 6 & 7 \\
\hline
i & 5 \\
\end{array}
\]

\[
\begin{array}{ccccccc}
0 & 2 & 4 & 4 & 6 & 6 & 5 \\
\hline
i & 7 \\
\end{array}
\]

**swap** shown in the lecture about lists
Insertion Sort: Moving into Position

\[
i = 0 \\
\text{while } i < n: \\
\quad \text{push_down}(b, i) \\
\quad i = i + 1 \\
\]

\[
def \text{push_down}(b, i):
\quad j = i \\
\quad \text{while } j > 0:
\quad \quad \text{if } b[j - 1] > b[j]:
\quad \quad \quad \text{swap}(b, j - 1, j) \\
\quad \quad j = j - 1
\]

\[
\begin{array}{ll|l|l}
0 & i & 0 & i \\
2 & 4 & 4 & 6 & 6 & 7 & 5 \\
2 & 4 & 4 & 6 & 5 & 6 & 7 \\
2 & 4 & 4 & 6 & 5 & 6 & 7 \\
\end{array}
\]
**Insertion Sort: Moving into Position**

\[ i = 0 \]

\[ \text{while } i < n: \]

\[ \text{push\_down}(b, i) \]

\[ i = i + 1 \]

\[ \text{def push\_down}(b, i): \]

\[ j = i \]

\[ \text{while } j > 0: \]

\[ \text{if } b[j-1] > b[j]: \]

\[ \text{swap}(b, j-1, j) \]

\[ j = j - 1 \]

\[ 0 \quad 2 \quad 4 \quad 4 \quad 6 \quad 6 \quad 7 \]

\[ i \]

\[ 5 \]

\[ 0 \quad 2 \quad 4 \quad 4 \quad 6 \quad 6 \quad 5 \]

\[ i \]

\[ 7 \]

\[ 0 \quad 2 \quad 4 \quad 4 \quad 6 \quad 5 \quad 6 \]

\[ i \]

\[ 7 \]

\[ 0 \quad 2 \quad 4 \quad 4 \quad 5 \quad 6 \quad 6 \]

\[ i \]

\[ 7 \]
The Importance of Helper Functions

i = 0
while i < n:
    push_down(b,i)
i = i+1

def push_down(b, i):
    j = i
    while j > 0:
        if b[j-1] > b[j]:
            swap(b,j-1,j)
        j = j-1

VS

Can you understand all this code below?

i = 0
while i < n:
    j = i
    while j > 0:
        if b[j-1] > b[j]:
            temp = b[j]
            b[j] = b[j-1]
            b[j-1] = temp
        j = j -1
i = i +1
Measuring Performance

- Performance is a tricky thing to measure
  - Different computers run at different speeds
  - Memory also has a major effect as well
- Need an independent way to measure
  - Measure in terms of “basic steps”
  - **Example**: Searching counted # of checks
- For sorting, we measure in terms of **swaps**
  - Three assignment statements
  - Present in all sorting algorithms
def push_down(b, i):
    """Push value at position i into sorted position in b[0..i-1]""
    j = i
    while j > 0:
        if b[j-1] > b[j]:
            swap(b,j-1,j)
        j = j-1
        Total Swaps: $0 + 1 + 2 + 3 + \ldots + (n-1) = \frac{(n-1)n}{2} = \frac{n^2-n}{2}$

• b[0..i-1]: i elements

• Worst case:
  - i = 0: 0 swaps
  - i = 1: 1 swap
  - i = 2: 2 swaps

• Pushdown is in a loop
  - Called for i in 0..n
  - i swaps each time
**Insertion Sort: Performance**

```python
def push_down(b, i):
    """Push value at position i into sorted position in b[0..i-1]""
    j = i
    while j > 0:
        if b[j-1] > b[j]:
            swap(b, j-1, j)
        j = j-1
```

- **b[0..i-1]:** i elements
- **Worst case:**
  - i = 0: 0 swaps
  - i = 1: 1 swap
  - i = 2: 2 swaps
- **Pushdown is in a loop**
  - Called for i in 0..n
  - i swaps each time

**Total Swaps:** $0 + 1 + 2 + 3 + \ldots + (n-1) = \frac{(n-1)n}{2} = \frac{n^2-n}{2}$

*Insertion sort is an $n^2$ algorithm*
Algorithm “Complexity”

- **Given**: a list of length $n$ and a problem to solve
- **Complexity**: *rough* number of steps to solve worst case
- Suppose we can compute 1000 operations a second:

<table>
<thead>
<tr>
<th>Complexity</th>
<th>n=10</th>
<th>n=100</th>
<th>n=1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\log n$</td>
<td>0.003 s</td>
<td>0.006 s</td>
<td>0.01 s</td>
</tr>
<tr>
<td>$n$</td>
<td>0.01 s</td>
<td>0.1 s</td>
<td>1 s</td>
</tr>
<tr>
<td>$n \log n$</td>
<td>0.016 s</td>
<td>0.32 s</td>
<td>4.79 s</td>
</tr>
<tr>
<td>$n^2$</td>
<td>0.1 s</td>
<td>10 s</td>
<td>16.7 m</td>
</tr>
<tr>
<td>$n^3$</td>
<td>1 s</td>
<td>16.7 m</td>
<td>11.6 d</td>
</tr>
<tr>
<td>$2^n$</td>
<td>1 s</td>
<td>$4 \times 10^{19}$ y</td>
<td>$3 \times 10^{290}$ y</td>
</tr>
</tbody>
</table>
Algorithm “Complexity”

- **Given**: a list of length n and a problem to solve
- **Complexity**: *rough* number of steps to solve worst case
- Suppose we can compute 1000 operations a second:

<table>
<thead>
<tr>
<th>Complexity</th>
<th>n=10</th>
<th>n=100</th>
<th>n=1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>log n</td>
<td></td>
<td>0.006 s</td>
<td>0.01 s</td>
</tr>
<tr>
<td>Linear Search</td>
<td></td>
<td>0.1 s</td>
<td>1 s</td>
</tr>
<tr>
<td>n log n</td>
<td>0.016s</td>
<td>0.32 s</td>
<td>4.79 s</td>
</tr>
<tr>
<td>n^2</td>
<td></td>
<td>10 s</td>
<td>16.7 m</td>
</tr>
<tr>
<td>n^3</td>
<td>1 s</td>
<td>16.7 m</td>
<td>11.6 d</td>
</tr>
<tr>
<td>2^n</td>
<td>1 s</td>
<td>4x10^{19} y</td>
<td>3x10^{290} y</td>
</tr>
</tbody>
</table>
Algorithm “Complexity”

- **Given**: a list of length n and a problem to solve
- **Complexity**: *rough* number of steps to solve worst case
- Suppose we can compute 1000 operations a second:

<table>
<thead>
<tr>
<th>Complexity</th>
<th>n=10</th>
<th>n=100</th>
<th>n=1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>log n</td>
<td>0.003 s</td>
<td>0.036 s</td>
<td>0.01 s</td>
</tr>
<tr>
<td>n</td>
<td>0.01 s</td>
<td>1 s</td>
<td>10 s</td>
</tr>
<tr>
<td>n log n</td>
<td>0.016 s</td>
<td>0.32 s</td>
<td>4.79 s</td>
</tr>
<tr>
<td>n^2</td>
<td>0.1 s</td>
<td>10 s</td>
<td>16.7 m</td>
</tr>
<tr>
<td>n^3</td>
<td>1 s</td>
<td>16.7 m</td>
<td>11.6 d</td>
</tr>
<tr>
<td>2^n</td>
<td>1 s</td>
<td>4x10^{19} y</td>
<td>3x10^{290} y</td>
</tr>
</tbody>
</table>

Major Topic in 2110: Beyond scope of this course
Insertion Sort is Not Great

• Typically $n^2$ is okay, but not great
  ▪ Will perform horribly on large data
  ▪ Very bad when performance critical (games)

• We would like to do better than this
  ▪ Can we get $n$ swaps (no)?
  ▪ How about $n \log n$ (maybe)

• This will require a new algorithm
  ▪ Let’s return to horizontal notation
A New Algorithm

Start: \( b \)

Goal: \( b \) sorted

In-Progress: \( b \) sorted, \( \leq b[i..] \) \( \geq b[0..i-1] \)

First segment always contains smaller values
Selection Sort

\[ i = 0 \]

while \( i < n \):

\[
\begin{array}{c|c|c|c}
0 & i & n \\
\hline
\text{sorted, } \leq b[i..] & \geq b[0..i-1] & \\
\end{array}
\]

# Find minimum in \( b[i..] \)

# Move it to position \( i \)

\( i = i + 1 \)

Remember the restrictions!
Selection Sort

How fast is this?

\[ i = 0 \]
\[ \text{while } i < n: \]
\[ \quad j = \text{index of min of } b[i..n-1] \]
\[ \quad \text{swap}(b, i, j) \]
\[ i = i + 1 \]
Selection Sort

i = 0

while i < n:
    j = index of min of b[i..n-1]
    swap(b,i,j)

This is also $n^2$!

i = i+1

This is $n$ steps
What is the Problem

• Both insertion, selection sort are nested loops
  ▪ Outer loop over each element to sort
  ▪ Inner loop to put next element in place
  ▪ Each loop is n steps. $n \times n = n^2$

• To do better we must *eliminate* a loop
  ▪ But with what? Recursion!

• But to do this we have to back up a bit
  ▪ Need to introduce an intermediate algorithm
The Problem Statement

- Given a list $b[h..k]$ with some value $x$ in $b[h]$:

  \[
  \begin{array}{c|c|c}
    h & x & ? \\
  \end{array}
  \]

- Start: $b \leq x \ x \geq x$

- Swap elements of $b[h..k]$ to get this answer:

  \[
  \begin{array}{c|c|c|c|c|c|c}
    h & i & i+1 & k \\
  \end{array}
  \]

- Goal: $b \leq x \ x \geq x$

- In-Progress: $b \leq x \ x \ ? \geq x$

Indices $b, h$ important!
Might partition only part
Partition Algorithm

- Given a list segment b[h..k] with some value x in b[h]:

  [\[
  \begin{array}{c|c}
  \text{h} & \text{k} \\
  \hline
  \text{Start: } & \begin{array}{c|c}
  \text{b} & \begin{array}{c}
  \text{x} \\
  \end{array} \\
  \end{array}
  \end{array}
  \]

- Swap elements of b[h..k] to get this answer

  [\[
  \begin{array}{c|c|c}
  \text{h} & \text{i} & \text{i+1} & \text{k} \\
  \hline
  \text{Goal: } & \begin{array}{c|c|c}
  \text{b} & \begin{array}{c}
  \text{\leq x} \\
  \text{x} \\
  \text{\geq x}
  \end{array}
  \end{array}
  \end{array}
  \]

change:

  [\[
  \begin{array}{c|c|c|c}
  \text{h} & \text{i} & \text{i+1} & \text{k} \\
  \hline
  \text{b} & \begin{array}{c}
  \begin{array}{c}
  \text{3} \ 5 \ 4 \ 1 \ 6 \ 2 \ 3 \ 8 \ 1
  \end{array}
  \end{array}
  \end{array}
  \]

into

  [\[
  \begin{array}{c|c|c|c}
  \text{h} & \text{i} & \text{i+1} & \text{k} \\
  \hline
  \text{b} & \begin{array}{c}
  \begin{array}{c}
  1 \ 2 \ 1 \ 3 \ 5 \ 4 \ 6 \ 3 \ 8
  \end{array}
  \end{array}
  \end{array}
  \]

or

  [\[
  \begin{array}{c|c|c|c}
  \text{h} & \text{i} & \text{i+1} & \text{k} \\
  \hline
  \text{b} & \begin{array}{c}
  \begin{array}{c}
  1 \ 2 \ 3 \ 1 \ 3 \ 4 \ 5 \ 6 \ 8
  \end{array}
  \end{array}
  \end{array}
  \]

- x is called the pivot value
  - x is not a program variable
  - denotes value initially in b[h]
def partition(b, h, k):

    """Partition list b[h..k] around a pivot x = b[h]"""
    i = h; j = k+1; x = b[h]

    while i < j-1:
        if b[i+1] >= x:
            # Move to end of block.
            swap(b,i+1,j-1)
            j = j - 1
        else:   # b[i+1] < x
            swap(b,i,i+1)
            i = i + 1

    return i

partition(b,h,k), not partition(b[h:k+1])

Remember, slicing always copies the list!

We want to partition the original list
def partition(b, h, k):
    # Partition list b[h..k] around a pivot x = b[h]
    i = h; j = k+1; x = b[h]

    while i < j-1:
        if b[i+1] >= x:
            # Move to end of block.
            swap(b,i+1,j-1)
            j = j - 1
        else:
            # b[i+1] < x
            swap(b,i,i+1)
            i = i + 1

    return i
def partition(b, h, k):
    """Partition list b[h..k] around a pivot x = b[h]"""
    i = h; j = k+1; x = b[h]

    while i < j-1:
        if b[i+1] >= x:
            # Move to end of block.
            swap(b,i+1,j-1)
            j = j - 1
        else:   # b[i+1] < x
            swap(b,i,i+1)
            i = i + 1

    return i
def partition(b, h, k):
    """Partition list b[h..k] around a pivot x = b[h]"""
    i = h; j = k+1; x = b[h]
    while i < j-1:
        if b[i+1] >= x:
            # Move to end of block.
            swap(b,i+1,j-1)
            j = j - 1
        else:   # b[i+1] < x
            swap(b,i,i+1)
            i = i + 1
    return i
def partition(b, h, k):
    """Partition list b[h..k] around a pivot x = b[h]"""
    i = h; j = k+1; x = b[h]

    while i < j-1:
        if b[i+1] >= x:
            # Move to end of block.
            swap(b, i+1, j-1)
            j = j - 1
        else:  # b[i+1] < x
            swap(b, i, i+1)
            i = i + 1

    return i
Why is this Useful?

- Will use this algorithm to replace inner loop
  - The inner loop cost us n swaps every time
- Can this reduce the number of swaps?
  - Worst case is k-h swaps
  - This is n if partitioning the whole list
  - But less if only partitioning part
- **Idea:** Break up list and partition only part?
  - This is *Divide-and-Conquer*!
Sorting with Partitions

• Given a list segment $b[h..k]$ with some value $x$ in $b[h]$:  

\[
\begin{array}{c}
\text{h} \\
\mid \text{x} \mid \\
\text{?} \\
\text{k}
\end{array}
\]

Start:

\[
\begin{array}{c}
\text{b} \\
\mid \text{<= x} \mid \text{x} \mid \text{>= x} \\
\text{k}
\end{array}
\]

Goal:  

• Swap elements of $b[h..k]$ to get this answer

Recursive partitions = sorting

- Called **QuickSort** (why???)
- Popular, fast sorting technique
Sorting with Partitions

- Given a list segment $b[h..k]$ with some value $x$ in $b[h]$:

  \[ \begin{array}{cc}
  h & k \\
  \text{Start: } b & \begin{array}{ccc} x & ? \\
  \end{array} \\
  \end{array} \]

- Swap elements of $b[h..k]$ to get this answer

  \[ \begin{array}{cccc}
  h & i & i+1 & k \\
  \text{Goal: } b & \begin{array}{cccc} y & ? & x & \geq x \\
  \end{array} \\
  \end{array} \]

Partition Recursively

Recursive partitions = sorting
- Called **QuickSort** (why???)
- Popular, fast sorting technique
Sorting with Partitions

- Given a list segment $b[h..k]$ with some value $x$ in $b[h]$:
  
  **Start:** $b[x]$

  **Goal:** $b[\leq y \ y \ >\ y \ x \ >\ x]$

- Swap elements of $b[h..k]$ to get this answer

Recursive partitions = sorting
- Called **QuickSort** (why???)
- Popular, fast sorting technique
QuickSort

```python
def quick_sort(b, h, k):
    """Sort the array fragment b[h..k]"""
    if b[h..k] has fewer than 2 elements:
        return
    j = partition(b, h, k)
    # b[h..j–1] <= b[j] <= b[j+1..k]
    # Sort b[h..j–1] and b[j+1..k]
    quick_sort(b, h, j-1)
    quick_sort(b, j+1, k)
```

- **Worst Case:**
  - array already sorted
  - Or almost sorted
  - $n^2$ in that case
- **Average Case:**
  - array is scrambled
  - $n \log n$ in that case
  - Best sorting time!

<table>
<thead>
<tr>
<th>h</th>
<th>i</th>
<th>i+1</th>
<th>k</th>
</tr>
</thead>
<tbody>
<tr>
<td>pre:</td>
<td>b</td>
<td>x</td>
<td>?</td>
</tr>
<tr>
<td>post:</td>
<td>b</td>
<td>&lt;= x</td>
<td>x</td>
</tr>
</tbody>
</table>
So Does that Solve It?

• Worst case still seems bad! Still $n^2$
  • Only happens in small number of cases
  • Just happens that case is common (already sorted)
• Can greatly reduce issue with randomization
  • Swap start with random element in list
  • Now pivot is random and already sorted unlikely

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>x</td>
<td>?</td>
<td>y</td>
</tr>
</tbody>
</table>

Start:
So Does that Solve It?

- Worst case still seems bad! Still $n^2$
  - Only happens in small number of cases
  - Just happens that case is common (already sorted)
- Can greatly reduce issue with randomization
  - Swap start with random element in list
  - Now pivot is random and already sorted unlikely

![Diagram]

Start: $b \ [x \ ? \ y \ ?]$
Can We Do Better?

• There is guaranteed n log n sorting algorithm
  ▪ Called **merge sort** (beyond scope of course)
  ▪ Used heavily in large databases
  ▪ But it has high overhead (slower on small data)

• What does the `sort()` method use?
  ▪ Uses **Timsort** (invented by Tim Peters in 2002)
  ▪ Combination of insertion sort and merge sort
  ▪ Insertion on small data, merge sort on large
Can We Do Better?

- There is guaranteed $n \log n$ sorting algorithm
  - Called **merge sort** (beyond scope of course)
  - Used heavily in large databases
  - But it has high overhead (slower on small data)
- What does the `sort()` method use?
  - Uses **Timsort** (invented by Tim Peters in 2002)
  - Combination of insertion sort and merge sort
  - Insertion on small data, 

Quicksort is 1959!