Searching and Sorting

• We all spend too much time searching for things and sorting through our possessions

• Computers do, too
  – Google
    – traverses web pages, keeping track of information
    – sorts info in some order for easier searching later

Many Kinds of Searches

• Looking in a collection for a specific value (e.g., looking for your name in a list of prize winners).
• Looking for a value with a specific property (e.g., finding the object on a computer screen containing the mouse location).
• Looking for a record in a database (e.g., finding all CDs by a given artist).
• Searching for all occurrences of a value in a collection (e.g., finding all occurrences of the word “search” in your Java text).

Will focus on the first.

Linear Searches of Arrays

• Many reasons to search an array for a specific value
  – Ex. Array of racers, in order of finish time;
    Search for a specific racer to determine her place.
• Will use arrays of ints as our examples
• int variable key specifies item to find

Recursive Solution

• Array not a recursively defined structure
• Can still search recursively
  – extra parameter to indicate where to start

for-loop selection

• look through array from beginning
• if item found, return index
• if not found, return -1 (not a valid index)

// Search for key in array. Return the first index of key, or -1 if it is not found.
public int search( int[] array, int key ) {
  for ( int index = 0; index < array.length; index++ ) {
    if ( key == array[index] ) {
      return index;
    }
  }
  return -1;
}
Comparable Versions

• Prefer to have same parameters for calls to recursive and non-recursive searches
• User should not have to worry about which implementation it is
• Provide extra method

```java
// Search for key in array. Return the first index of key, or // -1 if it is not found.
public int recSearchStart( int[] array, int key ) {
    return recSearch( array, key, 0 );
}
```

Predicting Timing of a Task

• Often useful to predict how long a task will take
• Problems with predicting exact time
  – will get different results on different computers
  – large data sets different from small
  – Finding element at beginning of search different from finding at end
• Prefer measures that predict relative time
  – For searches, count number of comparisons with key
  – consider "worst-case" behavior
  – "average-case" behavior

Complexity of Linear Search

For iterative search of array
• worst case
  – key found at end or not at all
  – n comparisons if array has n elements
• Average case
  – assumptions
    • key occurs exactly once in array
    • key element equally likely in any position
  – on avg, found halfway through array, i.e., n/2

Order Makes Searching Faster

• Can find elements with fewer comparisons if array is in order
• Average-case analysis
  – can be extended for cases where key not in array
  – stop when item larger than key found
  – average comparisons still n/2

Taking Advantage of Order

• Say you need to look up someone’s number in a telephone book
  – Would never consider using linear search
  – Take advantage of what you know about order of names in book
• Can take advantage of ordering in array searches in a clever way

Guessing Games

• I’m thinking of a number between 1 and 7.
  – You guess 4
  – I say too low
  – You guess 6
  – Too high
  – Must be 5!
• Each guess eliminates half of the remaining options
• Use this idea for search
Binary Search

- check middle element to check for key in array
- if element higher than key
  - look for key in middle of lower range
- else
  - look for key in middle of higher range

A divide and conquer strategy: divide a problem into smaller pieces

Recursive Binary Search

- starter method will make initial call of binarySearch method
- start and end will specify start and end indices of array segment to search
- middleIndex calculated from start and end

Tracing through a Search

Say key is 22
- start = 0, end = 14, middleIndex = 7
- 22 < array[7], so binarySearch(array, 22, 0, 6)
- start = 0, end = 6, middleIndex = 3
- 22 > array[3], so binarySearch( array 22, 4, 6)
- start = 4, end = 6, middleIndex = 5
- 22 = array[5], so Done!

Iterative Binary Search

- parameters start and end replaced with local variables
- testing whether any elements left now done in condition of while loop

Time Complexity of Binary Search

- 7 items in arrays; worst case
  - check middle item of 7
  - check middle item of remaining 3
  - check final 1 Item

- 31 items; worst case:
  - check middle item of 31
  - check middle item of remaining 15
  - check middle item of remaining 7
  - check middle item of remaining 3
  - check final item

- Half of remaining items eliminated with each check
Worst Case Complexity
If number of items in array is \( n \)
then need at most \( \log_2(n+1) \) guesses

Comparing Efficiency

<table>
<thead>
<tr>
<th>( n )</th>
<th>linear (unsorted)</th>
<th>binary (sorted)</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>15</td>
<td>15</td>
<td>8</td>
</tr>
<tr>
<td>31</td>
<td>31</td>
<td>10</td>
</tr>
<tr>
<td>127</td>
<td>127</td>
<td>14</td>
</tr>
<tr>
<td>1023</td>
<td>1027</td>
<td>20</td>
</tr>
<tr>
<td>1,048,575</td>
<td>1,056,783</td>
<td>40</td>
</tr>
</tbody>
</table>

Sorting
• Will develop algorithms for arrays of int
  – easy to change for other types of elements
• Resulting order will be nondecreasing
  – easy to change for nonincreasing

Selection Sort
• Simple way to think about sorting
  – Find smallest element and put in first available slot
  – Find second smallest; put in next slot
  – Find third smallest; put in next slot
• Want to use as little extra space as possible
  – perform sort in place
  – move data around in array; no need for second array

Swapping
• Many algorithms require ability to interchange values of two variables
• to sort in place, will need to swap values in array[i] and array[j]

private void swap( int[] array, int i, int j, ) {
    int temp = array[i];
    array[i] = array[j];
    array[j] = temp;
}

Why is temp needed?
Iterative Selection Sort

• Keep track of slotToFill
  – initially 0
  – then 1, 2, etc.
• Find index of smallest element in segment of array not yet sorted
• Swap smallest element with element at index slotToFill
• Value of slotToFill during final execution of for loop is array.length-2 why?

Operation of Selection Sort

Array of 5 ints
index: 0 1 2 3 4
elts[index]: 48 62 38 51 15
• slotToFill is 0
  – Find index of smallest in elts[0..4]
  – smallest in elts[4]
  – swap elts[4] with elts[0]

index: 0 1 2 3 4
elts[index]: 15 62 38 51 48
• slotToFill is 1
  – Find index of smallest in elts[1..4]
  – smallest in elts[2]

index: 0 1 2 3 4
elts[index]: 15 38 62 51 48
• slotToFill is 2
  – Find index of smallest in elts[2..4]
  – smallest in elts[4]

index: 0 1 2 3 4
elts[index]: 15 38 48 51 62
• slotToFill is 3
  – Find index of smallest in elts[3..4]
  – smallest in elts[3]

When slotToFill is incremented again, it will be equal to 4, which is array.length-1, so the for loop will terminate and elts is now in order

index: 0 1 2 3 4
elts[index]: 15 38 48 51 62
Recursive Selection Sort

To sort elements in array beginning at index start
• if fewer than 2 elements in array beginning at start
  – then all done!
• otherwise
  – find smallest element in portion of array beginning with start
  – swap smallest with item at start
  – selection sort remaining elements, i.e., beginning at index
    start + 1

```java
public void sort( int[] array ) {
    recursiveSelectionSort( array, 0 );
}
// PRE: startIndex must be a valid index for array
// POST: array[startIndex..array.length-1] has been rearranged and
//       is in nondecreasing order.
private void recursiveSelectionSort( int[] array, int startIndex ) {
    if (startIndex < array.length - 1) {
        // Find smallest element in rest of array
        int smallest = indexOfSmallest( array, startIndex );
        // Move smallest to index startIndex
        swap( array, smallest, startIndex );
        // Sort everything in the array after startIndex
        recursiveSelectionSort( array, startIndex + 1 );
    }
}
```
Insertion

- Need to find appropriate position for insertion
- May need to shift elements to make room for new one

Recursive Insertion Sort

For a collection of n elements
- If there are 1 or fewer items, stop
- Otherwise
  - perform insertion sort of first n-1 elements
  - insert nth element in order

Complexity of Insertion Sort: Worst Case

- insertNext called for 2-element array, 3-element,...n-element array
- yields 1 comparison + 2 + ... + (n-1)
- (n-1)n/2 as for selection sort

Average Case Complexity

- comparisons done to find position for insertion
- can stop comparing as soon as position found
  - on average, half way through array
- Total approximately n(n-1)/4

Faster Sorting

- Can we do better than n(n-1)/2?
- Try divide-and-conquer strategy
- Observe
  - in recursive selection sort
    - found smallest item
    - did selection sort of remaining n-1 items
  - in recursive insertion sort
    - did insertion sort of n-1 elements
    - inserted final element
- Recall from searches
  - Recursive calls on half as many items more efficient than recursive calls on one less item

Merge Sort

- if fewer than 2 elements to sort, all done!
- Else
  - recursively sort elements in first half of array
  - recursively sort elements in second half
  - merge two sorted halves
Merging Sorted Sublists

For two sorted sublists in same array:
index: 0 1 2 3 4 5 6 7 8 9
elts[index]: 15 38 48 51 62 7 12 23 49 88

Merge into new array, sorted
index: 0 1 2 3 4 5 6 7 8 9
sorted[index]: 7 12 15 23 38 48 49 51 62 88

Performing a Merge

index: 0 1 2 3 4 5 6 7 8 9
eelts[index]: 15 38 48 51 62 7 12 23 49 88

• compare first element of each half
• smallest (elts[5]) assigned to sorted[0]

index: 0 1 2 3 4 5 6 7 8 9
sorted[index]: 7 - - - - - - - - -

index: 0 1 2 3 4 5 6 7 8 9
eelts[index]: 15 38 48 51 62 7 12 23 49 88

• compare leftmost unplaced elements of each half
• smallest (elts[6]) assigned to sorted[1]

index: 0 1 2 3 4 5 6 7 8 9
sorted[index]: 7 12 - - - - - - -

index: 0 1 2 3 4 5 6 7 8 9
eelts[index]: 15 38 48 51 62 7 12 23 49 88

• compare leftmost unplaced elements
• smallest (elts[0]) assigned to sorted[2]

index: 0 1 2 3 4 5 6 7 8 9
sorted[index]: 7 12 15 - - - - -

A merge Method

• move array elements to tempArray first
  – a “helper” array
• merge elements back into original
• indexLeft and indexRight keep track of
  leftmost unmerged elements
• after one half used up, transfer rest of other
  half without comparison

Recursive Merge Sort

public void sort( int[] array ) {
  int[] tempArray = new int[array.length];
  mergeSort( array, 0, array.length-1, tempArray );
}

// PRE: left and right are valid indices of array with left <= right - 1.
// tempArray.length == array.length
// POST: elts of array[left..right] are rearranged in nondecreasing order
public void mergeSort( int[] array, int left, int right,
int[] tempArray ) {
  if (left < right) {
    int middle = (right + left) / 2
    mergeSort( array, left, middle, tempArray );
    mergeSort( array, middle + 1, right, tempArray );
    merge( array, left, middle, right, tempArray );
  }
}
**Complexity of MergeSort**

- Count comparisons: all performed in merge
- Merging two halves with total of \( k \) elements takes at most \( k-1 \) comparisons
- Let \( T(n) \) be a function that gives number of comparisons on array of size \( n \) in worst case
  - \( T(0) = 0, T(1) = 0, T(2) = 1 \)
  - \( T(n) \leq T(n/2) + T(n/2) + (n-1) \)
  - Simplify to
    \[ T(n) = 2 \cdot T(n/2) + n \]

**Worst Case Complexity**

But what, exactly, is the complexity?

\[
T(n) = n \times \log_2 n
\]

---

**Impact of Complexity**

*Average* comparisons to sort array of size \( n \):

<table>
<thead>
<tr>
<th>( n )</th>
<th>Selection</th>
<th>Insertion</th>
<th>Merge</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>190</td>
<td>116</td>
<td>64</td>
</tr>
<tr>
<td>40</td>
<td>780</td>
<td>416</td>
<td>166</td>
</tr>
<tr>
<td>100</td>
<td>4,950</td>
<td>2,565</td>
<td>542</td>
</tr>
<tr>
<td>1,000</td>
<td>499,500</td>
<td>245,132</td>
<td>8,709</td>
</tr>
</tbody>
</table>