Searching and Sorting

Most problems involve searching and sorting
Searching

The problem:
• You have a dataset, stored in some data structure
• You want to find a particular element:
  • If the element is in the dataset, return it (or its index)
  • If the element is NOT in the dataset, know so for sure
Searching: Linear Search

- A linear search examines each element one at a time, until it is either found or not found because it is not in the dataset.

- Elements may appear in any order in the dataset (i.e. does not assume items are sorted).

- Easy to implement, not very efficient!
public static Comparable linearSearch (Comparable[] data, Comparable target)
{
    Comparable result = null;
    int index = 0;

    while (result == null && index < data.length) {
        if (data[index].compareTo(target) == 0)
            result = data[index];
        index++;
    }

    return result;
} //What’s the complexity?
Searching: Binary Search

- If the elements are sorted and the dataset is indexable, we can be more efficient

- A binary search eliminates large parts of the search pool with each comparison

1. Begin searching at the middle
2. If the target isn’t found there, it may be in one half or the other
3. Repeat by searching in the appropriate half

```
3  5  12 19 50 52 60 60 85 91 92 93 98
```
public static Comparable binarySearch(Comparable[] data, Comparable target) {

    Comparable result = null;
    int first = 0, last = data.length - 1, mid;

    while (result == null && first <= last) {
        mid = (first + last) / 2;  // determine midpoint
        if (data[mid].compareTo(target) == 0)  // found target
            result = data[mid];
        else {
            if (data[mid].compareTo(target) > 0)  // in first half
                last = mid - 1;
            else
                first = mid + 1;
        }
    }

    return result;
} // What's the complexity?
Big-O complexity of binary search?

- Space to search is halved each step
- How many times can you halve \( n \) to get to 1?

- Remember: it only works if the input is already sorted!
Given a dataset with Comparable data.

You want to sort that data (increasing or decreasing order).

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>91</td>
<td>50</td>
<td>12</td>
<td>60</td>
<td>5</td>
<td>52</td>
</tr>
<tr>
<td>3</td>
<td>19</td>
<td>85</td>
<td>60</td>
<td>98</td>
<td>93</td>
</tr>
<tr>
<td>92</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>5</td>
<td>12</td>
<td>19</td>
<td>50</td>
<td>52</td>
</tr>
<tr>
<td>60</td>
<td>60</td>
<td>85</td>
<td>91</td>
<td>92</td>
<td>93</td>
</tr>
<tr>
<td>98</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Sorting using selectionSort**

- *SelectionSort* orders values by repeatedly putting a particular value into its final sorted position

- The algorithm
  1. find the smallest value in the dataset
  2. switch it with the value in the first position
  3. find the next smallest value in the dataset
  4. switch it with the value in the second position
  5. repeat until all values are in their sorted positions

Visualization [https://visualgo.net/sorting](https://visualgo.net/sorting)
public static void selectionSort (Comparable[] data) {
    int min;
    for(int index = 0; index < data.length - 1; index++) {
        min = index;
        for(int scan = index + 1; scan < data.length; scan++)
            if(data[scan].compareTo(data[min]) < 0)
                min = scan;
        swap(data, min, index);
    }
}
Sorting using insertionSort

- *InsertionSort* orders a list of values by repeatedly inserting a particular value into a sorted subset of the list.

- The algorithm
  1. consider the first item to be a sorted sublist of length 1
  2. insert the second item in the sorted sublist, shifting the first item if needed
  3. insert the third item into the sorted sublist, shifting the other items as needed
  4. repeat until all values inserted into their proper positions

(3) is sorted sublist. Consider 9.
Shift nothing, **insert 9**. (3, 9) are sorted. Consider 6.
Shift 9, **insert 6**. (3, 6, 9) are sorted. Consider 1.
Shift 3, 6, 9, **insert 1**. (1, 3, 6, 9) sorted. Consider 2.
Shift 3, 6, 9, **insert 2**. (1, 2, 3, 6, 9) sorted.
public static void insertionSort(Comparable[] data) {
    for (int index = 1; index < data.length; index++) {
        Comparable key = data[index];
        int position = index;

        //shift larger values to the right
        while (position > 0 && data[position - 1].compareTo(key) > 0) {
            data[position] = data[position - 1];
            position--;
        }
        data[position] = key;
    }
} //What’s the complexity?
Sorting using mergeSort

- Merge sort orders a list of values by recursively dividing the list in half until each sublist has one element, then recombining (merging) the sublists

- The algorithm

**Decomposition Step**
1. divide the list into two (roughly) equal parts
2. recursively divide each part in half, continuing until a part contains only one element

**Merging Step**
1. merge the two parts into one sorted list
2. continue to merge parts as the recursion unfolds
Merge sort – Decomposition step
Merge sort – Merging step

```
90  65  7
   |   |   \
  90  7   65
   |   |   \
  7   65  90
```

```
305 120
   |   |   \
  120 305
   |   |   \
   8  110
```

```
110  8
   |   |   \
  110  8
   |   |   \
   110
```

```
7   8   65   90   110   120   305
```

public static void mergeSort(Comparable[] data, int min, int max) {
    if (min < max) {
        int mid = (min + max) / 2;
        mergeSort(data, min, mid);
        mergeSort(data, mid+1, max);
        merge(data, min, mid, max);
    }
}

WARNING: This is NOT the exact code executing in the visualization of the previous slide
public static void merge(Comparable[] data, int first, int mid, int last) {
    Comparable[] temp = new Comparable[data.length];

    int first1 = first, last1 = mid; // endpoints for 1st subarray
    int first2 = mid + 1, last2 = last; // endpoints for 2nd subarray
    int index = first1; // next index open in temp array

    // copy smaller item from each subarray into temp
    // until one of the subarrays is exhausted
    while (first1 <= last1 && first2 <= last2) {
        if (data[first1].compareTo(data[first2]) < 0) {
            temp[index] = data[first1];
            first1++;
        } else {
            temp[index] = data[first2];
            first2++;
        }
        index++;
    }
    // more...
merge (cont.)

//copy remaining elements from first subarray, if any
while(first1 <= last1){
    temp[index] = data[first1];
    first1++;
    index++;
}

//copy remaining elements from second subarray, if any
while(first2 <= last2){
    temp[index] = data[first2];
    first2++;
    index++;
}

//copy merged data into original array
for(index = first; index <= last; index++){
    data[index] = temp[index];
}
}
Quick sort orders a list of values by partitioning the list around some element (the pivot), then sorting each partition.

The algorithm:
1. choose one element in the list to be the partition element
2. organize the elements so that all elements less than the partition element are to the left and all elements greater than the partition element are to the right
3. apply quickSort (recursively) to each partition

Nice if the partition element divides the list roughly in half.

Quick sort has two methods:
- quickSort – performs recursive algorithm
- partition – rearranges elements into two partitions
Example of Quicksort

- Input array
- First element becomes the pivot
- Pivot is placed to its final position during partition
- Two sub-arrays are sorted recursively
- First elements become pivots
- Pivots are placed in final position
- Four sub-arrays are sorted recursively...
public static void quickSort(
    Comparable[] data, int min, int max) {
    int pivot;

    if(min < max){
        pivot = partition(data, min, max); //make partitions
        quickSort(data, min, pivot-1); //sort left partition
        quickSort(data, pivot+1, max); //sort right partition
    }
}
Example of partition

Input array

First element becomes the pivot

Elements are exchanged to separate smaller from larger

Pivot is exchanged to its final position

Two sub-arrays are sorted recursively
private static int partition(Comparable[] data, int min, int max) {

    Comparable partitionValue = data[min];
    int left = min; int right = max;

    while (left < right) {
        // search for an element that is > the partition element
        while (data[left].compareTo(partitionValue) <= 0 && left < right)
            left++;

        // search for an element that is < the partition element
        while (data[right].compareTo(partitionValue) > 0)
            right--;

        if (left < right)
            swap(data, left, right);
    }

    // move the partition element to its final position
    swap(data, min, right);

    return right; // will become the pivot
## Sort Efficiencies
### Array Sorting Algorithms

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Time Complexity</th>
<th>Space Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Best</td>
<td>Average</td>
</tr>
<tr>
<td>Quicksort</td>
<td>$\Omega(n \log(n))$</td>
<td>$\Theta(n \log(n))$</td>
</tr>
<tr>
<td>Mergesort</td>
<td>$\Omega(n \log(n))$</td>
<td>$\Theta(n \log(n))$</td>
</tr>
<tr>
<td>Timsort</td>
<td>$\Omega(n)$</td>
<td>$\Theta(n \log(n))$</td>
</tr>
<tr>
<td>Heapsort</td>
<td>$\Omega(n \log(n))$</td>
<td>$\Theta(n \log(n))$</td>
</tr>
<tr>
<td>Bubble Sort</td>
<td>$\Omega(n)$</td>
<td>$\Theta(n^2)$</td>
</tr>
<tr>
<td>Insertion Sort</td>
<td>$\Omega(n)$</td>
<td>$\Theta(n^2)$</td>
</tr>
<tr>
<td>Selection Sort</td>
<td>$\Omega(n^2)$</td>
<td>$\Theta(n^2)$</td>
</tr>
<tr>
<td>Tree Sort</td>
<td>$\Omega(n \log(n))$</td>
<td>$\Theta(n \log(n))$</td>
</tr>
<tr>
<td>Shell Sort</td>
<td>$\Omega(n \log(n))$</td>
<td>$\Theta(n(\log(n))^2)$</td>
</tr>
<tr>
<td>Bucket Sort</td>
<td>$\Omega(n+k)$</td>
<td>$\Theta(n+k)$</td>
</tr>
<tr>
<td>Radix Sort</td>
<td>$\Omega(nk)$</td>
<td>$\Theta(nk)$</td>
</tr>
<tr>
<td>Counting Sort</td>
<td>$\Omega(n+k)$</td>
<td>$\Theta(n+k)$</td>
</tr>
<tr>
<td>Cubesort</td>
<td>$\Omega(n)$</td>
<td>$\Theta(n \log(n))$</td>
</tr>
</tbody>
</table>