Searching and Sorting

Searching

The problem:
- You have a set of data,
- You want to find a particular element in that set of data,
  - If the element is in the data, return it and/or its index
  - If the element is NOT in the data, know so for sure
A linear search examines each element one at a time, until it is either found or every element has been examined.

- Elements may be in any order (i.e. does not assume items are sorted)
- Easy to implement, not very efficient!

```
start

```

```java
//Search specified array using linear search.
//Returns null if target not found
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;
}
```
Searching: Binary Search

- **Assume the elements are sorted**, now 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, if it exists it is in one half or the other
  3. Recurse to the middle of that half, and begin again

```
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
            return result;
        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: to use it, the input has to already be sorted!)

![Binary Search Diagram](image)

Sorting

- You have some Comparable data.
- You want to sort that data.

![Sorting Diagram](image)
**Sorting algorithm:**

**Selection Sort**

- *Selection Sort* orders values by repeatedly putting a particular value into its final position

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

Visualization [https://visualgo.net/sorting](https://visualgo.net/sorting)

```java
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);
    }
}
//What’s the complexity?
```
### Sorting algorithm: Insertion Sort

- **Insertion sort** 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.

```java
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: Merge Sort

- Merge sort orders a list of values by recursively dividing the list in half until each sublist has one element, then recombining.
- 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 - Merging step

```
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++;
  }

  //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];
  }
Sorting: Quick sort

- Quick sort orders a list of values by partitioning the list around one 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 greater are to the right
  3. apply the quick sort algorithm (recursively) to both partitions

- 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...
quickSort

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>Ω(n log(n))</td>
<td>θ(n log(n))</td>
</tr>
<tr>
<td>Mergesort</td>
<td>Ω(n log(n))</td>
<td>θ(n log(n))</td>
</tr>
<tr>
<td>Timsort</td>
<td>Ω(n)</td>
<td>θ(n log(n))</td>
</tr>
<tr>
<td>Heapsort</td>
<td>Ω(n log(n))</td>
<td>θ(n log(n))</td>
</tr>
<tr>
<td>Bubble Sort</td>
<td>Ω(n)</td>
<td>θ(n log(n))</td>
</tr>
<tr>
<td>Insertion Sort</td>
<td>Ω(n)</td>
<td>θ(n log(n))</td>
</tr>
<tr>
<td>Selection Sort</td>
<td>Ω(n log(n))</td>
<td>θ(n log(n))</td>
</tr>
<tr>
<td>Tree Sort</td>
<td>Ω(n log(n))</td>
<td>θ(n log(n))</td>
</tr>
<tr>
<td>Shell Sort</td>
<td>Ω(n log(n))</td>
<td>θ(n log(n))</td>
</tr>
<tr>
<td>Bucket Sort</td>
<td>Ω(n+k)</td>
<td>θ(n+k)</td>
</tr>
<tr>
<td>Radix Sort</td>
<td>Ω(nk)</td>
<td>θ(nk)</td>
</tr>
<tr>
<td>Counting Sort</td>
<td>Ω(n+k)</td>
<td>θ(n+k)</td>
</tr>
<tr>
<td>Cubesort</td>
<td>Ω(n)</td>
<td>θ(n log(n))</td>
</tr>
</tbody>
</table>