# Searching and Sorting

**October 19, 2018**

- Searching
- Sorting
- Complexity

# The Problem

**1: Searching**
**1.1: The Problem**

Problem: Find an entry in a collection of entries

Algorithm: Search an array for a given key

- Input: An array of `Comparable`s and a `Comparable` key
- Output: Index into array or a `NotFound` exception

# Comparables

**Aside 1**

A `Comparable` type is a type with a total ordering — every element is less than, equal to, or greater than every other element. This is determined by the `compareTo` method. If `a` and `b` are `Comparables` of the same type:

- If $a < b$ then `a.compareTo(b) < 0`
- If $a = b$ then `a.compareTo(b) = 0`
- If $a > b$ then `a.compareTo(b) > 0`

See, for example, `Strings`.

# Sequential Search

## 1.2: Sequential Search

### 1.2.1: Description

Here we simply work through the array element by element until we find an occurrence of the key or we run out of array to search.

### 1.2.2: Pseudocode

```
for (int index = 0; index < array length; index++) {
    if (index^th element of array == our key) {
        return with the current index;
    }
}
throw a NotFound exception;
```

# Sequential Search

### 1.2.3: Java Code

```
public <T extends Comparable<? super T>> int
     search (T key,T[] list) throws NotFound {
   for (int i = 0; i < list.length; i++) {
      if (list[i].compareTo(key) == 0) {
         return i;
      }
   }
   throw new NotFound();
}
```

# Comparables

**Aside 2**

Why

```
<T extends Comparable<? super T>> int search
```

1. `T extends Comparable` is required to ensure `T` is comparable.
2. `Comparable` is a generic class. The generic type specifies the type of object that may be compared to.

   ```
   class MyClass implements Comparable<Integer>
   ```

3. `T extends Comparable<T> int search` is too restrictive — requires that `compareTo` is defined for `T`
4. `T extends Comparable<? super T> int search` allows for `compareTo` to be defined for `T`, *or some superclass of* `T`

# Binary Search

**1.3: Binary Search**

**1.3.1: Description**

A “divide and conquer” algorithm — only works if the list is sorted.

**1.3.2: Binary Search**

```
if (element in middle of array is required element) {
    return element's index;
} else if (middle element is bigger than required element)
    search lower half of array;
} else {
    search upper half of array;
}
```

# Binary Search

**1.3.3: Examples**
**1.3.3 A: Example 1**

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **−62** | **−27** | **−15** | **−13** | **−5** | **0** | **3** | **8** | **9** | **12** | **17** | **29** | **32** | **54** |

Find **−5**

# Binary Search

**1.3.3: Examples**
**1.3.3 A: Example 1**

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $-62$ | $-27$ | $-15$ | $-13$ | $-5$ | $0$ | $3$ | $8$ | $9$ | $12$ | $17$ | $29$ | $32$ | $54$ |

$$-5 < 3$$

# Binary Search

**1.3.3: Examples**
**1.3.3 A: Example 1**

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $-62$ | $-27$ | $-15$ | $-13$ | $-5$ | $0$ | $3$ | $8$ | $9$ | $12$ | $17$ | $29$ | $32$ | $54$ |

$$-15 < -5$$

# Binary Search

**1.3.3: Examples**

**1.3.3 A: Example 1**

| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
| $-62$ | $-27$ | $-15$ | $-13$ | $-5$ | $0$ | $3$ | $8$ | $9$ | $12$ | $17$ | $29$ | $32$ | $54$ |

$-5 = -5$: **return** index 4

# Binary Search

**1.3.3: Examples**
**1.3.3 B: Example 2**

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| **2** | **5** | **8** | **12** | **26** | **45** | **51** | **72** |

Find **37**

# Binary Search

**1.3.3: Examples**
**1.3.3 B: Example 2**

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| **2** | **5** | **8** | **12** | **26** | **45** | **51** | **72** |

$\mathbf{12} < \mathbf{37}$

# Binary Search

**1.3.3: Examples**
**1.3.3 B: Example 2**

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 2 | 5 | 8 | 12 | **26** | **45** | **51** | **72** |

$$\mathbf{37} < \mathbf{45}$$

# Binary Search

**1.3.3: Examples**
**1.3.3 B: Example 2**

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 2 | 5 | 8 | 12 | 26 | 45 | 51 | 72 |

$\mathbf{26} < \mathbf{37}$

# Binary Search

**1.3.3: Examples**
**1.3.3 B: Example 2**

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 2 | 5 | 8 | 12 | 26 | 45 | 51 | 72 |

Key could not be found

# Binary Search

### 1.3.4: Java Code

```java
/**
* Top level call: starts recursive search
*/

public <T extends Comparable<? super T>>
    int search(T key,T[] list) throws NotFound {
   return searchBetween(0,list.length-1,key,list);
}
```

# Binary Search

```
public <T extends Comparable<? super T>>
    int searchBetween(int bottom,int top, T Key,T[] list)
      throws notFound {
   int pivot = (bottom + top)/2;
   if (list[pivot].compareTo(key) == 0) {
      return pivot;
   } else if (top == bottom) {
      throw new NotFound();
   } else if (list[pivot].compareTo(key) > 0) {
      return searchBetween(bottom,pivot-1,key,list);
   } else {
      return searchBetween(pivot+1,top,key,list);
   }
}
```

Another example

# Problem

**2: Sorting**
**2.1: Introduction**
**2.1.1: The Problem**

Imagine we have a pack of playing cards and (for simplicity) we will assume that we have only got the suit of diamonds.
The aim is to sort the diamond suit.

# Solutions

**2.1.2: Solutions**

- sort the pile we have been given (*in situ* sorting),
- or move cards from our unsorted pile to a new sorted pile ("copy" sorting).

# Solution

We will maintain two piles of cards:

- the one given to us,
- and a sorted pile (obviously, this is initially empty)

Generally we want to sort *in situ*. In in situ sorting the "two piles" are both parts of the original pile — e.g. the "sorted pile" might be the first **n** elements of the pile.

# Bubble Sort

## 2.2: Bubble Sort

### 2.2.1: Description

The sorted part of the list is the last part of the list.

```
for (the whole list is unsorted;
   there is still part of the list unsorted;
   one more element has been sorted) {
   for (elements to be compared = first two elements;
      we haven't reached the end of the unsorted part;
      select the next pair of elements to be compared) {
      if (the first element > the second element) {
         swap them;
      }
   }
}
```

# Bubble Sort

## 2.2.2: Algorithm

```
public <T extends Comparable<? super T>>
    void bubbleSort(T[] array) {
  for (int unsorted = array.length-1; uns'd > 0; uns'd--)
    for (int nextToCompare = 0; // start with first two elements
      nextToCompare < lastUnsorted; // stop at end of unso
      nextToCompare++) {
      if (array[nextToCompare].compareTo(array[nextToCom
        // the elements in wrong order so swap them
        swap(array,nextToCompare,nextToCompare+1);
      }
    }
  }
}
```

# Selection Sort

**2.3: Selection Sort**

Similar to bubble sort, but reduce the number of swaps by simply finding the *largest* element in the unsorted part and swapping it with the *last* element in the unsorted part.

The sorted part of the list is the last part of the list

```
for (the whole list is unsorted;
   there is still part of the list unsorted;
   another element has been sorted) {
   find the (index of) largest unsorted element;
   swap it with the last unsorted element;
}
```

# Selection Sort

**2.3.1: Algorithm**
The algorithm is left as an exercise.

# Insertion Sort

## 2.4: Insertion Sort
### 2.4.1: Description

- The sorted part of the list is the first part of the list
- **for** 
```
(the whole list is unsorted;
   part of the list is still unsorted;
   another element is sorted) {
   take the first unsorted element;
   insert it into the correct position in
     the sorted part;
}
```

# Insertion Sort

## 2.4.2: Algorithm

```
public <T extends Comparable<? super T>>
     void insertionSort(T[] array) {
    for (sorted = 0; sorted < array.length-1; sorted++) {
       T newElement = array[sorted+1];
       int compareTo = sorted; // start by comparing last sorted
       while (compareTo >= 0 &&
            newElement.compareTo(array[compareTo]) < 0) {
          array[compareTo+1] = array[compareTo];
          compareTo--;
       }
       array[compareTo+1] = newElement;
    }
}
```

## 2.5: Merge Sort

### 2.5.1: Description

Merge sort is the basic “divide and conquer” algorithm.

- If the length of the list is less than two it is sorted
- Otherwise split it into two halves and apply merge sort to these two sublists
- Now merge the two lists maintaining the ordering.

# Merge Sort

**2.5.2: Algorithm**
**2.5.2 A: Merging**

```
public <T extends Comparable<? super T>>
    void merge(T[] target,T[] source1,T[] source2) {
   int index1=0, index2=0;
   for (index = 0; index < target.length; index++) {
      if (source1[index1].compareTo(source2[index2]) < 0) {
         target[index] = source1[index1++];
      } else {
         target[index] = source2[index2++];
      }
   }
}
```

# Merge Sort

## 2.5.2 B: Copying

```
public <T> T[] copy(T[] source,int from,int to) {
    T[] copy = new T[to-from]; // can't actually do this
    for (int i = 0; i < copy.length; i++) {
        copy[i] = source[from+i];
    }
    return copy;
}
```

# Merge Sort

Can, e.g., do this. . .

```
public <T> T[] copy(T[] list,int from,int to) {
    List<T> copyL = new ArrayList<T>(to-from);
    for (int i = 0; i < to-from; i++) {
        copyL.add(i,list[from+i]);
    }
    return copyL.toArray(list);
}
```

# Merge Sort

### 2.5.2 C: Merge Sort

```
public <T extends Comparable<? super T>>
    void mergeSort(T[] array) {
   if (array.length < 2) {
      return;
   }
   T[] temp1 = copy(array,0,array.length/2-1);
   T[] temp2 = copy(array,array.length/2,array.length-1);
   mergeSort(temp1);
   mergeSort(temp2);
   merge(array,temp1,temp2);
}
```

# Quicksort

## 2.6: Quicksort
### 2.6.1: Description

- Split the list into two smaller, “more sorted” lists.
- Choose an element (called the “pivot”).
- Split the list into two sublists such that
  - all elements to the left of the pivot are smaller (or equal to) than the pivot
  - all elements to the right of the pivot are larger than the pivot

  Then the pivot will be in the right position!
- Repeat quicksort, separately, on the parts of the list to the left and right of the pivot.

# Splitting the list

**2.6.2: Splitting the List**

- Choose a pivot — e.g. the first element in the list.
- This leaves a gap in the list that will be to the left of the pivot.

. . .

# Splitting the list

. . .

**loop:** Find the last element in the list that is smaller than the pivot and appears after the gap.

- Put it in the gap.
- This leaves a gap towards the end of the list that will be to the right of the pivot.
- Find the first element in the list that is larger than the pivot and appears before the gap.
- Put it in the gap.
- Repeat from "**loop:**", continuing the search for small elements from where we left off. Stop when the "large element index" and the "small element index" meet. That's where the pivot goes.

Example

**3: Complexity**

Which sort/search algorithm to use?

**3.1: Sort**

- Bubble sort, selection sort and insertion sort are all, on average, slower than merge sort and quick sort. However they can all be written such that they are more efficient if the data is already (nearly) sorted — i.e. they can be efficient ways of checking if a list is already sorted.
- Merge sort and quicksort are both, on average, more efficient than the other algorithms. However merge sort requires copying the list — which can be space expensive.
- Quicksort is as bad as bubble sort, selection sort and insertion sort in the worst case — when the list is (nearly) sorted.

# Complexity

**3.2: Search**

Binary search is clearly more efficient than sequential search, but it will take longer to sort a random list than it will to do a sequential search of that list.

## 3.3: Conclusions

- If you know the list is sorted, use binary search
- If you know the list is unsorted
  - if you are only going to do one search, use sequential search
  - if you are going to do many searches, sort the list using, e.g., quicksort then use binary search
- If you think the list might be sorted, check with bubble, selection or insertion sort, then sort if necessary, then use binary search

# Complexity

E.g. for list size 100, with 10 searches:

| | List sorted | List unsorted |
|---|---|---|
| sequential search; | 1,000 | 1,000 |
| quicksort;<br>binary search; | 10,000 | 731 |
| **if** (!sorted) {<br>    quicksort;<br>}<br>binary search; | 166 | 831 |

# Complexity

| | No. of searches | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | 1 | | 10 | | 100 | | 1th | | 1m | |
| 10 | 10 | 10 | 100 | 100 | 1th | 1th | 10th | 10th | 10m | 10m |
| | 103 | 37 | 133 | 66 | 432 | 365 | 3.4th | 3.4th | 3.3m | 3.3m |
| | 13 | 47 | 43 | 76 | 342 | 375 | 3.3th | 3.4th | 3.3m | 3.3m |
| 100 | 100 | 100 | 1.0th | 1.0th | 10th | 10th | 100th | 100th | 100m | 100m |
| | 10th | 671 | 10th | 731 | 11th | 1.3th | 17th | 7.3th | 6.7m | 6.6m |
| | 107 | 771 | 166 | 831 | 764 | 1.4th | 6.7th | 7.4th | 6.6m | 6.6m |
| 1th | 1.0th | 1.0th | 10th | 10th | 100th | 100th | 1.0m | 1.0m | 1.0b | 1.0b |
| | 1.0m | 10th | 1.0m | 10th | 1.0m | 11th | 1.0m | 20th | 11m | 10m |
| | 1.0th | 11th | 1.1th | 11th | 2.0th | 12th | 11th | 21th | 10m | 10m |
| 1m | 1.0m | 1.0m | 10m | 10m | 100m | 100m | 1.0b | 1.0b | 1.0t | 1.0t |
| | 1.0t | 20m | 1.0t | 20m | 1.0t | 20m | 1.0t | 20m | 1.0t | 40m |
| | 1.0m | 21m | 1.0m | 21m | 1m | 21m | 1.0m | 21m | 21m | 41m |