Building Java Programs

Chapter 13
Lecture 13-1: binary search and complexity

reading: 13.1-13.2
Sequential search

- **sequential search**: Locates a target value in an array / list by examining each element from start to finish. Used in `indexOf`.
  
  - How many elements will it need to examine?
  
  - Example: Searching the array below for the value 42:

| index | 0  | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
|-------|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| value | -4 | 2  | 7  | 10 | 15 | 20 | 22 | 25 | 30 | 36 | 42 | 50 | 56 | 68 | 85 | 92 | 103|

- The array is sorted. Could we take advantage of this?
• **binary search**: Locates a target value in a *sorted* array or list by successively eliminating half of the array from consideration.

  • How many elements will it need to examine?
  
  • Example: Searching the array below for the value **42**:

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>-4</td>
<td>2</td>
<td>7</td>
<td>10</td>
<td>15</td>
<td>20</td>
<td>22</td>
<td>25</td>
<td>30</td>
<td>36</td>
<td>42</td>
<td>50</td>
<td>56</td>
<td>68</td>
<td>85</td>
<td>92</td>
<td>103</td>
</tr>
</tbody>
</table>

  min  |  mid | max
Arrays.binarySearch

// searches an entire sorted array for a given value
// returns its index if found; a negative number if not found
// Precondition: array is sorted
Arrays.binarySearch(array, value)

// searches given portion of a sorted array for a given value
// examines minIndex (inclusive) through maxIndex (exclusive)
// returns its index if found; a negative number if not found
// Precondition: array is sorted
Arrays.binarySearch(array, minIndex, maxIndex, value)

- The binarySearch method in the Arrays class searches an array very efficiently if the array is sorted.
  - You can search the entire array, or just a range of indexes (useful for "unfilled" arrays such as the one in ArrayIntList)
Using **binarySearch**

```java
// index    0  1  2  3  4  5  6  7  8  9  10  11  12  13  14  15
int[] a = {-4, 2, 7, 9, 15, 19, 25, 28, 30, 36, 42, 50, 56, 68, 85, 92};
int index  = Arrays.binarySearch(a, 0, 16, 42); // index1 is 10
int index2 = Arrays.binarySearch(a, 0, 16, 21); // index2 is -7
```

- **binarySearch** returns the index where the value is found
- if the value is *not* found, **binarySearch** returns: 
  
  
  ```java
  -(insertionPoint + 1)
  ```

  - where **insertionPoint** is the index where the element *would* have been, if it had been in the array in sorted order.
  - To insert the value into the array, negate **insertionPoint + 1**

```java
int indexToInsert21 = -(index2 + 1); // 6
```
Runtime Efficiency (13.2)

- How much better is binary search than sequential search?

- **efficiency**: measure of computing resources used by code.
  - can be relative to speed (time), memory (space), etc.
  - most commonly refers to run time

- Assume the following:
  - Any single Java statement takes same amount of time to run.
  - A method call's runtime is measured by the total of the statements inside the method's body.
  - A loop's runtime, if the loop repeats N times, is N times the runtime of the statements in its body.
Efficiency examples

statement1;
statement2;
statement3;  

\[
\text{for (int } i = 1; i \leq N; i++) \{ \\
\text{statement4; } \\
\}
\]

\[
\text{for (int } i = 1; i \leq N; i++) \{ \\
\text{statement5; } \\
\text{statement6; } \\
\text{statement7; } \\
\}
\]

\[
\begin{align*}
3 & \quad (3) \\
N & \quad (N) \\
3N & \quad (3N) \\
4N + 3 & \quad (4N + 3)
\end{align*}
\]
Efficiency examples 2

```java
for (int i = 1; i <= N; i++) {
    for (int j = 1; j <= N; j++) {
        statement1;
    }
}
```

```java
for (int i = 1; i <= N; i++) {
    statement2;
    statement3;
    statement4;
    statement5;
}
```

- How many statements will execute if \( N = 10 \)? If \( N = 1000 \)?
Algorithm growth rates (13.2)

- We measure runtime in proportion to the input data size, N.
  - **growth rate**: Change in runtime as N changes.

- Say an algorithm runs $0.4N^3 + 25N^2 + 8N + 17$ statements.
  - Consider the runtime when N is *extremely large*.
  - We ignore constants like 25 because they are tiny next to N.
  - The highest-order term ($N^3$) dominates the overall runtime.

- We say that this algorithm runs "on the order of" $N^3$.
- or $O(N^3)$ for short  ("Big-Oh of N cubed")
### Complexity classes

- **complexity class**: A category of algorithm efficiency based on the algorithm's relationship to the input size $N$.

<table>
<thead>
<tr>
<th>Class</th>
<th>Big-Oh</th>
<th>If you double $N$, ...</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>constant</td>
<td>$O(1)$</td>
<td>unchanged</td>
<td>10ms</td>
</tr>
<tr>
<td>logarithmic</td>
<td>$O(\log_2 N)$</td>
<td>increases slightly</td>
<td>175ms</td>
</tr>
<tr>
<td>linear</td>
<td>$O(N)$</td>
<td>doubles</td>
<td>3.2 sec</td>
</tr>
<tr>
<td>log-linear</td>
<td>$O(N \log_2 N)$</td>
<td>slightly more than doubles</td>
<td>6 sec</td>
</tr>
<tr>
<td>quadratic</td>
<td>$O(N^2)$</td>
<td>quadruples</td>
<td>1 min 42 sec</td>
</tr>
<tr>
<td>cubic</td>
<td>$O(N^3)$</td>
<td>multiplies by 8</td>
<td>55 min</td>
</tr>
<tr>
<td>exponential</td>
<td>$O(2^N)$</td>
<td>multiplies drastically</td>
<td>$5 \times 10^{61}$ years</td>
</tr>
</tbody>
</table>
Complexity classes

Sequential search

- What is its complexity class?

```java
public int indexOf(int value) {
    for (int i = 0; i < size; i++) {
        if (elementData[i] == value) {
            return i;
        }
    }
    return -1;  // not found
}
```

- On average, "only" N/2 elements are visited
  - 1/2 is a constant that can be ignored
Collection efficiency

- Efficiency of our `ArrayIntList` or Java's `ArrayList`:

<table>
<thead>
<tr>
<th>Method</th>
<th><code>ArrayList</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>O(1)</td>
</tr>
<tr>
<td>add(index, value)</td>
<td>O(N)</td>
</tr>
<tr>
<td>indexOf</td>
<td>O(N)</td>
</tr>
<tr>
<td>get</td>
<td>O(1)</td>
</tr>
<tr>
<td>remove</td>
<td>O(N)</td>
</tr>
<tr>
<td>set</td>
<td>O(1)</td>
</tr>
<tr>
<td>size</td>
<td>O(1)</td>
</tr>
</tbody>
</table>
Binary search

- **binary search** successively eliminates half of the elements.

  - **Algorithm:** Examine the middle element of the array.
    - If it is too big, eliminate the right half of the array and repeat.
    - If it is too small, eliminate the left half of the array and repeat.
    - Else it is the value we're searching for, so stop.

  - Which indexes does the algorithm examine to find value **42**?
  - What is the runtime complexity class of binary search?

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>-4</td>
<td>2</td>
<td>7</td>
<td>10</td>
<td>15</td>
<td>20</td>
<td>22</td>
<td>25</td>
<td>30</td>
<td>36</td>
<td>42</td>
<td>50</td>
<td>56</td>
<td>68</td>
<td>85</td>
<td>92</td>
<td>103</td>
</tr>
</tbody>
</table>

| min   |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |
| mid   |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |
| max   |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |
Binary search runtime

• For an array of size N, it eliminates $\frac{1}{2}$ until 1 element remains.
  
  N, N/2, N/4, N/8, ..., 4, 2, 1

• How many divisions does it take?

• Think of it from the other direction:
  
  • How many times do I have to multiply by 2 to reach N?
    
    1, 2, 4, 8, ..., N/4, N/2, N
  
  • Call this number of multiplications "x".

  \[2^x = N\]

  \[x = \log_2 N\]

• Binary search is in the \textbf{logarithmic} complexity class.
Sorting

- **sorting**: Rearranging the values in an array or collection into a specific order (usually into their "natural ordering").
  - one of the fundamental problems in computer science
  - can be solved in many ways:
    - there are many sorting algorithms
    - some are faster/slower than others
    - some use more/less memory than others
    - some work better with specific kinds of data
    - some can utilize multiple computers / processors, ...

- **comparison-based sorting**: determining order by comparing pairs of elements:
  - <, >, compareTo, ...
Sorting methods in Java

- The Arrays and Collections classes in java.util have a static method sort that sorts the elements of an array/list.

```java
String[] words = {"foo", "bar", "baz", "ball"};
Arrays.sort(words);
System.out.println(Arrays.toString(words));
// [ball, bar, baz, foo]

List<String> words2 = new ArrayList<String>();
for (String word : words) {
    words2.add(word);
}
Collections.sort(words2);
System.out.println(words2);
// [ball, bar, baz, foo]
```
Sorting algorithms

- **bogo sort**: shuffle and pray
- **bubble sort**: swap adjacent pairs that are out of order
- **selection sort**: look for the smallest element, move to front
- **insertion sort**: build an increasingly large sorted front portion
- **merge sort**: recursively divide the array in half and sort it
- **heap sort**: place the values into a sorted tree structure
- **quick sort**: recursively partition array based on a middle value

Other specialized sorting algorithms:
- **bucket sort**: cluster elements into smaller groups, sort them
- **radix sort**: sort integers by last digit, then 2nd to last, then ...
- ...
Selection sort

- **selection sort**: Orders a list of values by repeatedly putting the smallest or largest unplaced value into its final position.

The algorithm:
- Look through the list to find the smallest value.
- Swap it so that it is at index 0.
- Look through the list to find the second-smallest value.
- Swap it so that it is at index 1.
  ...
- Repeat until all values are in their proper places.
Selection sort example

- **Initial array:**

| index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
|-------|---|---|---|---|---|---|---|---|---|---|-----|-----|-----|-----|-----|-----|
| value | 22| 18| 12| -4| 27| 30| 36| 50| 7 | 68| 91 | 56 | 2 | 85 | 42 | 98 | 25 |

- **After 1st, 2nd, and 3rd passes:**

| index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
|-------|---|---|---|---|---|---|---|---|---|---|-----|-----|-----|-----|-----|-----|
| value | -4| 18| 12| **22**| 27| 30| 36| 50| 7 | 68| 91 | 56 | 2 | 85 | 42 | 98 | 25 |

| index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
|-------|---|---|---|---|---|---|---|---|---|---|-----|-----|-----|-----|-----|-----|
| value | -4| **2**| 12| 22 | 27| 30| 36| 50| 7 | 68| 91 | 56 | **18**| 85 | 42 | 98 | 25 |

| index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
|-------|---|---|---|---|---|---|---|---|---|---|-----|-----|-----|-----|-----|-----|
| value | -4| 2 | **7**| 22 | 27| 30| 36| 50| **12**| 68| 91 | 56 | 18 | 85 | 42 | 98 | 25 |
Selection sort code

// Rearranges the elements of a into sorted order using
// the selection sort algorithm.
public static void selectionSort(int[] a) {
    for (int i = 0; i < a.length - 1; i++) {
        // find index of smallest remaining value
        int min = i;
        for (int j = i + 1; j < a.length; j++) {
            if (a[j] < a[min]) {
                min = j;
            }
        }
        // swap smallest value its proper place, a[i]
        swap(a, i, min);
    }
}
Selection sort runtime (Fig. 13.6)

- What is the complexity class (Big-Oh) of selection sort?

<table>
<thead>
<tr>
<th>N</th>
<th>Runtime (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>0</td>
</tr>
<tr>
<td>2000</td>
<td>16</td>
</tr>
<tr>
<td>4000</td>
<td>47</td>
</tr>
<tr>
<td>8000</td>
<td>234</td>
</tr>
<tr>
<td>16000</td>
<td>657</td>
</tr>
<tr>
<td>32000</td>
<td>2562</td>
</tr>
<tr>
<td>64000</td>
<td>10265</td>
</tr>
<tr>
<td>128000</td>
<td>41141</td>
</tr>
<tr>
<td>256000</td>
<td>164985</td>
</tr>
</tbody>
</table>
Similar algorithms

- **bubble sort**: Make repeated passes, swapping adjacent values
  - slower than selection sort (has to do more swaps)

- **insertion sort**: Shift each element into a sorted sub-array
  - faster than selection sort (examines fewer values)
Merge sort

- **merge sort**: Repeatedly divides the data in half, sorts each half, and combines the sorted halves into a sorted whole.

The algorithm:
- Divide the list into two roughly equal halves.
- Sort the left half.
- Sort the right half.
- Merge the two sorted halves into one sorted list.

- Often implemented recursively.
- An example of a "divide and conquer" algorithm.
  - Invented by John von Neumann in 1945
Merge sort example

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>22</td>
<td>18</td>
<td>12</td>
<td>-4</td>
<td>58</td>
<td>7</td>
<td>31</td>
<td>42</td>
</tr>
</tbody>
</table>

```
merge
split
22 18 12 -4
split
22 18
split
22
merge
18 22
merge
-4 12
merge
-4 12 18 22
merge
-4 7 12 18 22 31 42 58
```
## Merging sorted halves

<table>
<thead>
<tr>
<th>Subarrays</th>
<th>Next include</th>
<th>Merged array</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3</td>
<td>0 1 2 3</td>
<td>0 1 2 3 4 5 6 7</td>
</tr>
<tr>
<td>14 32 67 76</td>
<td>23 41 58 85</td>
<td>14 23 32 41 58 67 76 85</td>
</tr>
</tbody>
</table>

### Merging process:
1. **Merging sorted halves**: Start by comparing the first elements of each subarray. The smallest element is added to the merged array and the index of the subarray from which it was taken is incremented.
2. **Continue the process**: Repeat the comparison and merging until all elements from both subarrays are included in the merged array.
3. **Final result**: The merged array contains all elements from both subarrays in sorted order.

### Example:
- **Initial subarrays**: 14 32 67 76 and 23 41 58 85
- **Merging process**:
  - 14 is the smallest, added to the merged array: 14
  - 23 is the next smallest, added to the merged array: 14 23
  - ... 
  - **Final merged array**: 14 23 32 41 58 67 76 85
Merge halves code

// Merges the left/right elements into a sorted result.
// Precondition: left/right are sorted
public static void merge(int[] result, int[] left,
                          int[] right) {
  int i1 = 0;  // index into left array
  int i2 = 0;  // index into right array

  for (int i = 0; i < result.length; i++) {
    if (i2 >= right.length ||
        (i1 < left.length && left[i1] <= right[i2])) {
      result[i] = left[i1];  // take from left
      i1++;
    } else {
      result[i] = right[i2];  // take from right
      i2++;
    }
  }
}
// Rearranges the elements of a into sorted order using
// the merge sort algorithm.
public static void mergeSort(int[] a) {
    // split array into two halves
    int[] left = Arrays.copyOfRange(a, 0, a.length/2);
    int[] right = Arrays.copyOfRange(a, a.length/2, a.length);

    // sort the two halves
    ...

    // merge the sorted halves into a sorted whole
    merge(a, left, right);
}
// Rearranges the elements of a into sorted order using
// the merge sort algorithm (recursive).
public static void mergeSort(int[] a) {
    if (a.length >= 2) {
        // split array into two halves
        int[] left = Arrays.copyOfRange(a, 0, a.length/2);
        int[] right = Arrays.copyOfRange(a, a.length/2, a.length);

        // sort the two halves
        mergeSort(left);
        mergeSort(right);

        // merge the sorted halves into a sorted whole
        merge(a, left, right);
    }
}
Merge sort runtime

What is the complexity class (Big-Oh) of merge sort?

<table>
<thead>
<tr>
<th>N</th>
<th>Runtime (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>0</td>
</tr>
<tr>
<td>2000</td>
<td>0</td>
</tr>
<tr>
<td>4000</td>
<td>0</td>
</tr>
<tr>
<td>8000</td>
<td>0</td>
</tr>
<tr>
<td>16000</td>
<td>0</td>
</tr>
<tr>
<td>32000</td>
<td>15</td>
</tr>
<tr>
<td>64000</td>
<td>16</td>
</tr>
<tr>
<td>128000</td>
<td>47</td>
</tr>
<tr>
<td>256000</td>
<td>125</td>
</tr>
<tr>
<td>512000</td>
<td>250</td>
</tr>
<tr>
<td>1e6</td>
<td>532</td>
</tr>
<tr>
<td>2e6</td>
<td>1078</td>
</tr>
<tr>
<td>4e6</td>
<td>2265</td>
</tr>
<tr>
<td>8e6</td>
<td>4781</td>
</tr>
<tr>
<td>1.6e7</td>
<td>9828</td>
</tr>
<tr>
<td>3.3e7</td>
<td>20422</td>
</tr>
<tr>
<td>6.5e7</td>
<td>42406</td>
</tr>
<tr>
<td>1.3e8</td>
<td>88344</td>
</tr>
</tbody>
</table>
Bogo sort

- **bogo sort**: Orders a list of values by repetitively shuffling them and checking if they are sorted.
  - name comes from the word "bogus"

The algorithm:
- Scan the list, seeing if it is sorted. If so, stop.
- Else, shuffle the values in the list and repeat.

- This sorting algorithm (obviously) has terrible performance!
  - What is its runtime?
Bogo sort code

// Places the elements of a into sorted order.
public static void bogoSort(int[] a) {
    while (!isSorted(a)) {
        shuffle(a);
    }
}

// Returns true if a's elements are in sorted order.
public static boolean isSorted(int[] a) {
    for (int i = 0; i < a.length - 1; i++) {
        if (a[i] > a[i + 1]) {
            return false;
        }
    }
    return true;
}
Bogo sort code, cont'd.

// Shuffles an array of ints by randomly swapping each
// element with an element ahead of it in the array.
public static void shuffle(int[] a) {
    for (int i = 0; i < a.length - 1; i++) {
        // pick a random index in [i+1, a.length-1]
        int range = a.length - 1 - (i + 1) + 1;
        int j = (int) (Math.random() * range + (i + 1));
        swap(a, i, j);
    }
}

// Swaps a[i] with a[j].
public static void swap(int[] a, int i, int j) {
    if (i != j) {
        int temp = a[i];
        a[i] = a[j];
        a[j] = temp;
    }
}