Building Java Programs

Chapter 13
binary search and complexity

reading: 13.1-13.2
Tips for testing

- You cannot test every possible input, parameter value, etc.
  - Think of a limited set of tests likely to expose bugs.

- Think about boundary cases
  - Positive; zero; negative numbers
  - Right at the edge of an array or collection's size

- Think about empty cases and error cases
  - 0, -1, null; an empty list or array

- Test behavior in combination
  - Maybe `add` usually works, but fails after you call `remove`
  - Make multiple calls; maybe `size` fails the second time only
Searching methods

- Implement the following methods:
  - `indexOf` - returns first index of element, or -1 if not found
  - `contains` - returns true if the list contains the given int value

- Why do we need `isEmpty` and `contains` when we already have `indexOf` and `size`?
  - Adds convenience to the client of our class:

```java
// less elegant                   // more elegant
if (myList.size() == 0) {
  if (myList.isEmpty()) {
    if (myList.indexOf(42) >= 0) {
      if (myList.contains(42)) {
```
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**:

<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>

- The array is sorted. Could we take advantage of this?
Binary search (13.1)

- **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>

\[\text{index} \quad 0 \quad 1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8 \quad 9 \quad 10 \quad 11 \quad 12 \quad 13 \quad 14 \quad 15 \quad 16 \]
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 \texttt{binarySearch}

\begin{verbatim}
// 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
\end{verbatim}

\begin{itemize}
  \item \texttt{binarySearch} returns the index where the value is found
  \item if the value is \textit{not} found, \texttt{binarySearch} returns: \[(\text{insertionPoint} + 1)\]
    \begin{itemize}
      \item \texttt{where insertionPoint} is the index where the element \textit{would} have been, if it had been in the array in sorted order.
      \item To insert the value into the array, negate \texttt{insertionPoint} + 1
    \end{itemize}
  \end{itemize}

\begin{verbatim}
int indexToInsert21 = -(index2 + 1);  // 6
\end{verbatim}
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

\[
\begin{align*}
\text{statement1;} \\
\text{statement2;} \\
\text{statement3;} \\
\{ & 3 \\
\text{for (int i = 1; i <= N; i++) {} } \\
\text{statement4;} \\
\} \\
\text{for (int i = 1; i <= N; i++) {} } \\
\text{statement5;} \\
\text{statement6;} \\
\text{statement7;} \\
\{ & 3N \\
\} \\
\{ & N \\
\} \\
\{ & 4N + 3 \\
\}
\end{align*}
\]
Efficiency examples 2

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

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

| 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|
Collection efficiency

- Efficiency of our ArrayIntList or Java's ArrayList:

<table>
<thead>
<tr>
<th>Method</th>
<th>ArrayList</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.
  \[ N, \frac{N}{2}, \frac{N}{4}, \frac{N}{8}, \ldots, 4, 2, 1 \]
  - 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 $N$?
    \[ 1, 2, 4, 8, \ldots, \frac{N}{4}, \frac{N}{2}, N \]
  - Call this number of multiplications "$x$".

  \[ 2^x = N \]
  \[ x = \log_2 N \]

- Binary search is in the **logarithmic** complexity class.
Range algorithm

What complexity class is this algorithm? Can it be improved?

// returns the range of values in the given array;
// the difference between elements furthest apart
// example: range({17, 29, 11, 4, 20, 8}) is 25
public static int range(int[] numbers) {
    int maxDiff = 0; // look at each pair of values
    for (int i = 0; i < numbers.length; i++) {
        for (int j = 0; j < numbers.length; j++) {
            int diff = Math.abs(numbers[j] - numbers[i]);
            if (diff > maxDiff) {
                maxDiff = diff;
            }
        }
    }
    return diff;
}
Range algorithm

What complexity class is this algorithm? Can it be improved?

// returns the range of values in the given array;
// the difference between elements furthest apart
// example: range({17, 29, 11, 4, 20, 8}) is 25
public static int range(int[] numbers) {
    int maxDiff = 0;  // look at each pair of values
    for (int i = 0; i < numbers.length; i++) {
        for (int j = 0; j < numbers.length; j++) {
            int diff = Math.abs(numbers[j] - numbers[i]);
            if (diff > maxDiff) {
                maxDiff = diff;
            }
        }
    }
    return diff;
}
Range algorithm 2

The last algorithm is $O(N^2)$. A slightly better version:

```java
// returns the range of values in the given array;
// the difference between elements furthest apart
// example: range({17, 29, 11, 4, 20, 8}) is 25
public static int range(int[] numbers) {
    int maxDiff = 0; // look at each pair of values
    for (int i = 0; i < numbers.length; i++) {
        for (int j = i + 1; j < numbers.length; j++) {
            int diff = Math.abs(numbers[j] - numbers[i]);
            if (diff > maxDiff) {
                maxDiff = diff;
            }
        }
    }
    return diff;
}
```
Range algorithm 3

This final version is $O(N)$. It runs MUCH faster:

```java
// returns the range of values in the given array;
// example: range({17, 29, 11, 4, 20, 8}) is 25
public static int range(int[] numbers) {
    int max = numbers[0]; // find max/min values
    int min = max;
    for (int i = 1; i < numbers.length; i++) {
        if (numbers[i] < min) {
            min = numbers[i];
        }
        if (numbers[i] > max) {
            max = numbers[i];
        }
    }
    return max - min;
}
```
Runtime of first 2 versions

• Version 1:

<table>
<thead>
<tr>
<th>N</th>
<th>Runtime (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>15</td>
</tr>
<tr>
<td>2000</td>
<td>47</td>
</tr>
<tr>
<td>4000</td>
<td>203</td>
</tr>
<tr>
<td>8000</td>
<td>781</td>
</tr>
<tr>
<td>16000</td>
<td>3110</td>
</tr>
<tr>
<td>32000</td>
<td>12563</td>
</tr>
<tr>
<td>64000</td>
<td>49937</td>
</tr>
</tbody>
</table>

• Version 2:

<table>
<thead>
<tr>
<th>N</th>
<th>Runtime (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>16</td>
</tr>
<tr>
<td>2000</td>
<td>16</td>
</tr>
<tr>
<td>4000</td>
<td>110</td>
</tr>
<tr>
<td>8000</td>
<td>406</td>
</tr>
<tr>
<td>16000</td>
<td>1578</td>
</tr>
<tr>
<td>32000</td>
<td>6265</td>
</tr>
<tr>
<td>64000</td>
<td>25031</td>
</tr>
</tbody>
</table>
**Runtime of 3rd version**

- **Version 3:**

<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>0</td>
</tr>
<tr>
<td>64000</td>
<td>0</td>
</tr>
<tr>
<td>128000</td>
<td>0</td>
</tr>
<tr>
<td>256000</td>
<td>0</td>
</tr>
<tr>
<td>512000</td>
<td>0</td>
</tr>
<tr>
<td>1e6</td>
<td>0</td>
</tr>
<tr>
<td>2e6</td>
<td>16</td>
</tr>
<tr>
<td>4e6</td>
<td>31</td>
</tr>
<tr>
<td>8e6</td>
<td>47</td>
</tr>
<tr>
<td>1.67e7</td>
<td>94</td>
</tr>
<tr>
<td>3.3e7</td>
<td>188</td>
</tr>
<tr>
<td>6.5e7</td>
<td>453</td>
</tr>
<tr>
<td>1.3e8</td>
<td>797</td>
</tr>
<tr>
<td>2.6e8</td>
<td>1578</td>
</tr>
</tbody>
</table>

![Graph showing runtime vs input size](image_url)
Max subsequence sum

• Write a method `maxSum` to find the largest sum of any contiguous subsequence in an array of integers.
  • Easy for all positives: include the whole array.
  • What if there are negatives?

<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>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>2</td>
<td>1</td>
<td>-4</td>
<td>10</td>
<td>15</td>
<td>-2</td>
<td>22</td>
<td>-8</td>
<td>5</td>
</tr>
</tbody>
</table>

Largest sum: 10 + 15 + -2 + 22 = 45

• (Let's define the max to be 0 if the array is entirely negative.)

• Ideas for algorithms?
Algorithm 1 pseudocode

maxSum(a):
    max = 0.
    for each starting index i:
        for each ending index j:
            sum = add the elements from a[i] to a[j].
            if sum > max,
                max = sum.
    return max.

<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>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>2</td>
<td>1</td>
<td>-4</td>
<td>10</td>
<td>15</td>
<td>-2</td>
<td>22</td>
<td>-8</td>
<td>5</td>
</tr>
</tbody>
</table>
Algorithm 1 code

- What complexity class is this algorithm?
- $O(N^3)$. Takes a few seconds to process 2000 elements.

```java
public static int maxSum1(int[] a) {
    int max = 0;
    for (int i = 0; i < a.length; i++) {
        for (int j = i; j < a.length; j++) {
            // sum = add the elements from a[i] to a[j].
            int sum = 0;
            for (int k = i; k <= j; k++) {
                sum += a[k];
            }
            if (sum > max) {
                max = sum;
            }
        }
    }
    return max;
}
```
Flaws in algorithm 1

- Observation: We are redundantly re-computing sums.
  - For example, we compute the sum between indexes 2 and 5:
  
  - Next we compute the sum between indexes 2 and 6:
  
  - We already had computed the sum of 2-5, but we compute it again as part of the 2-6 computation.

- Let's write an improved version that avoids this flaw.

<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>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>2</td>
<td>1</td>
<td>-4</td>
<td>10</td>
<td>15</td>
<td>-2</td>
<td>22</td>
<td>-8</td>
<td>5</td>
</tr>
</tbody>
</table>
Algorithm 2 code

- What complexity class is this algorithm?
- $O(N^2)$. Can process tens of thousands of elements per second.

```java
public static int maxSum2(int[] a) {
    int max = 0;
    for (int i = 0; i < a.length; i++) {
        int sum = 0;
        for (int j = i; j < a.length; j++) {
            sum += a[j];
            if (sum > max) {
                max = sum;
            }
        }
    }
    return max;
}
```

<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>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>2</td>
<td>1</td>
<td>-4</td>
<td>10</td>
<td>15</td>
<td>-2</td>
<td>22</td>
<td>-8</td>
<td>5</td>
</tr>
</tbody>
</table>
A clever solution

- **Claim 1**: A max range cannot start with a negative-sum range.

  
  \[
  \begin{array}{cccccc}
  i & \ldots & j & j+1 & \ldots & k \\
  \hline
  \text{< 0} & \text{sum}(j+1, k) \\
  \text{sum}(i, k) & \text{< sum}(j+1, k) \\
  \end{array}
  \]

- **Claim 2**: If \(\text{sum}(i, j-1) \geq 0\) and \(\text{sum}(i, j) < 0\), any max range that ends at \(j+1\) or higher cannot start at any of \(i\) through \(j\).

  
  \[
  \begin{array}{ccccccc}
  i & \ldots & j-1 & j & j+1 & \ldots & k \\
  \hline
  \geq 0 & < 0 & \text{sum}(j+1, k) \\
  < 0 & \text{sum}(j+1, k) \\
  \text{sum}(?, k) & \text{< sum}(j+1, k) \\
  \end{array}
  \]

- Together, these observations lead to a very clever algorithm...
Algorithm 3 code

- What complexity class is this algorithm?
  - \(O(N)\). Handles many millions of elements per second!

```java
public static int maxSum3(int[] a) {
    int max = 0;
    int sum = 0;
    int i = 0;
    for (int j = 0; j < a.length; j++) {
        if (sum < 0) {
            // if sum becomes negative, max range
            i = j; // cannot start with any of i - j-1
            sum = 0; // (Claim 2)
        }
        sum += a[j];
        if (sum > max) {
            max = sum;
        }
    }
    return max;
}
```