Building Java Programs

Chapter 13
Sorting

reading: 13.3, 13.4
Languages and Grammars
S

NP
John

VP
V
lost

NP
Det
his

N
pants
Languages and grammars

- **(formal) language**: A set of words or symbols.

- **grammar**: A description of a language that describes which sequences of symbols are allowed in that language.
  - describes language *syntax* (rules) but not *semantics* (meaning)
  - can be used to generate strings from a language, or to determine whether a given string belongs to a given language
Backus-Naur (BNF)

- **Backus-Naur Form (BNF):** A syntax for describing language grammars in terms of transformation *rules*, of the form:

  \[ \langle \text{symbol} \rangle ::= \langle \text{expression} \rangle \mid \langle \text{expression} \rangle \ldots \mid \langle \text{expression} \rangle \]

  - **terminal:** A fundamental symbol of the language.
  - **non-terminal:** A high-level symbol describing language syntax, which can be transformed into other non-terminal or terminal symbol(s) based on the rules of the grammar.

- developed by two Turing-award-winning computer scientists in 1960 to describe their new ALGOL programming language
An example BNF grammar

<s>::=<n> <v>
<n>::=Marty | Victoria | Stuart | Jessica
<v>::=cried | slept | belched

- Some sentences that could be generated from this grammar:
  
  Marty slept
  Jessica belched
  Stuart cried
BNF grammar version 2

\[ \text{s} ::= \text{np} \text{ v} \]
\[ \text{np} ::= \text{pn} \mid \text{dp} \text{ n} \]
\[ \text{pn} ::= \text{Marty} \mid \text{Victoria} \mid \text{Stuart} \mid \text{Jessica} \]
\[ \text{dp} ::= \text{a} \mid \text{the} \]
\[ \text{n} ::= \text{ball} \mid \text{hamster} \mid \text{carrot} \mid \text{computer} \]
\[ \text{v} ::= \text{cried} \mid \text{slept} \mid \text{belched} \]

- Some sentences that could be generated from this grammar:
  - the carrot cried
  - Jessica belched
  - a computer slept
BNF grammar version 3

<s>::=<np> <v>
<np>::=<pn> | <dp> <adj> <n>
<pn>::=Marty | Victoria | Stuart | Jessica
<dp>::=a | the
<adj>::=silly | invisible | loud | romantic
<n>::=ball | hamster | carrot | computer
<v>::=cried | slept | belched

- Some sentences that could be generated from this grammar:

  the invisible carrot cried
  Jessica belched
  a computer slept
  a romantic ball belched
Grammars and recursion

\[
\begin{align*}
&s ::= \langle np \rangle \hspace{1mm} \langle v \rangle \\
\langle np \rangle ::= & \langle pn \rangle | \langle dp \rangle \hspace{1mm} \langle adjp \rangle \hspace{1mm} \langle n \rangle \\
\langle pn \rangle ::= & \text{Marty} | \text{Victoria} | \text{Stuart} | \text{Jessica} \\
\langle dp \rangle ::= & \text{a} | \text{the} \\
\langle adjp \rangle ::= & \langle adj \rangle \hspace{1mm} \langle adjp \rangle | \langle adj \rangle \\
\langle adj \rangle ::= & \text{silly} | \text{invisible} | \text{loud} | \text{romantic} \\
\langle n \rangle ::= & \text{ball} | \text{hamster} | \text{carrot} | \text{computer} \\
\langle v \rangle ::= & \text{cried} | \text{slept} | \text{belched}
\end{align*}
\]

- Grammar rules can be defined \textit{recursively}, so that the expansion of a symbol can contain that same symbol.
- There must also be expressions that expand the symbol into something non-recursive, so that the recursion eventually ends.
Grammar, final version

<s>::=<np> <vp>
<np>::=<dp> <adjp> <n>|<pn>
<dp>::=the|a
<adjp>::=<adj>|<adj> <adjp>
<adj>::=big|fat|green|wonderful|faulty|subliminal
<n>::=dog|cat|man|university|father|mother|child
<pn>::=John|Jane|Sally|Spot|Fred|Elmo
<vp>::=<tv> <np>|<iv>
<tv>::=hit|honored|kissed|helped
<iv>::=died|collapsed|laughed|wept

- Could this grammar generate the following sentences?
  Fred honored the green wonderful child
  big Jane wept the fat man fat

- Generate a random sentence using this grammar.
Fred honored the green wonderful child
# Collections class

<table>
<thead>
<tr>
<th>Method name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>binarySearch(list, value)</code></td>
<td>returns the index of the given value in a sorted list (&lt; 0 if not found)</td>
</tr>
<tr>
<td><code>copy(listTo, listFrom)</code></td>
<td>copies <code>listFrom</code>'s elements to <code>listTo</code></td>
</tr>
<tr>
<td><code>emptyList()</code>, <code>emptyMap()</code>,</td>
<td>returns a read-only collection of the given type that has no elements</td>
</tr>
<tr>
<td><code>emptySet()</code></td>
<td></td>
</tr>
<tr>
<td><code>fill(list, value)</code></td>
<td>sets every element in the list to have the given value</td>
</tr>
<tr>
<td><code>max(collection)</code>, <code>min(collection)</code></td>
<td>returns largest/smallest element</td>
</tr>
<tr>
<td><code>replaceAll(list, old, new)</code></td>
<td>replaces an element value with another</td>
</tr>
<tr>
<td><code>reverse(list)</code></td>
<td>reverses the order of a list's elements</td>
</tr>
<tr>
<td><code>shuffle(list)</code></td>
<td>arranges elements into a random order</td>
</tr>
<tr>
<td><code>sort(list)</code></td>
<td>arranges elements into ascending order</td>
</tr>
</tbody>
</table>
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 ...
- ...
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;
    }
}

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:

<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>18</td>
<td>12</td>
<td>22</td>
<td>27</td>
<td>30</td>
<td>36</td>
<td>50</td>
<td>7</td>
<td>68</td>
<td>91</td>
<td>56</td>
<td>2</td>
<td>85</td>
<td>42</td>
<td>98</td>
<td>25</td>
</tr>
</tbody>
</table>

<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>12</td>
<td>22</td>
<td>27</td>
<td>30</td>
<td>36</td>
<td>50</td>
<td>7</td>
<td>68</td>
<td>91</td>
<td>56</td>
<td>18</td>
<td>85</td>
<td>42</td>
<td>98</td>
<td>25</td>
</tr>
</tbody>
</table>

<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>22</td>
<td>27</td>
<td>30</td>
<td>36</td>
<td>50</td>
<td>12</td>
<td>68</td>
<td>91</td>
<td>56</td>
<td>18</td>
<td>85</td>
<td>42</td>
<td>98</td>
<td>25</td>
</tr>
</tbody>
</table>
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>

Input size (N)
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.

- 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>

```
  22 18 12 -4
      /  \       /  \      /  \     /  \    /  \   /  \  /  \ /  \ /  \ /  \ /  \\
  22 18 | 12 -4
    /  \       /  \      /  \     /  \    /  \   /  \ /  \ /  \ /  \ /  \\
 22 18 | 12 -4
    /  \       /  \      /  \     /  \    /  \   /  \ /  \ /  \ /  \ /  \\
22 18 12 -4
      /  \       /  \      /  \     /  \    /  \   /  \ /  \ /  \ /  \ /  \\
  58 7 31 42
```
### Merging sorted halves

<table>
<thead>
<tr>
<th>Subarrays</th>
<th>Next include</th>
<th>Merged array</th>
</tr>
</thead>
<tbody>
<tr>
<td>14 32 67 76</td>
<td>23 41 58 85</td>
<td>14 from left</td>
</tr>
<tr>
<td>il</td>
<td>i2</td>
<td>i</td>
</tr>
<tr>
<td>14 32 67 76</td>
<td>23 41 58 85</td>
<td>23 from right</td>
</tr>
<tr>
<td>il</td>
<td>i2</td>
<td>i</td>
</tr>
<tr>
<td>14 32 67 76</td>
<td>23 41 58 85</td>
<td>32 from left</td>
</tr>
<tr>
<td>il</td>
<td>i2</td>
<td>i</td>
</tr>
<tr>
<td>14 32 67 76</td>
<td>23 41 58 85</td>
<td>41 from right</td>
</tr>
<tr>
<td>il</td>
<td>i2</td>
<td>i</td>
</tr>
<tr>
<td>14 32 67 76</td>
<td>23 41 58 85</td>
<td>58 from right</td>
</tr>
<tr>
<td>il</td>
<td>i2</td>
<td>i</td>
</tr>
<tr>
<td>14 32 67 76</td>
<td>23 41 58 85</td>
<td>67 from left</td>
</tr>
<tr>
<td>il</td>
<td>i2</td>
<td>i</td>
</tr>
<tr>
<td>14 32 67 76</td>
<td>23 41 58 85</td>
<td>76 from left</td>
</tr>
<tr>
<td>il</td>
<td>i2</td>
<td>i</td>
</tr>
<tr>
<td>14 32 67 76</td>
<td>23 41 58 85</td>
<td>85 from right</td>
</tr>
<tr>
<td>il</td>
<td>i2</td>
<td>i</td>
</tr>
</tbody>
</table>
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.

- An example of a "divide and conquer" algorithm.
  - Invented by John von Neumann in 1945
// 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);
    }
}
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>