Chapter 10
Recursion and Search
Recursion: General Overview

- **Recursion in Algorithms**
  - **Recursion** is the use of recursive algorithms to solve a problem
  - A recursive algorithm uses itself to solve one or more smaller identical problems

- **Recursion in Java**
  - A recursive method implements a recursive algorithm
  - A recursive method is a method that directly or indirectly makes a call to itself
Recursion: General Overview ...

- The recursion can be
  - **direct**: the method calls itself
  - **indirect**: the method calls another method that calls the first method
  - **single**: the method calls itself once
  - **Multiple**: the method calls itself multiple times
Recursion: General Overview ...

- Recursion does not refer to a circular logic
  - A recursive algorithm calls itself on a **different**, generally **simpler**, instance (i.e. with a **smaller input**)

```java
public void f(input){
    if (input < some_threshold) {
        // base case
        return;
    }
    // recursive case
    f(smaller_input);
}
```
Recursion: General Overview ...

public output f(input){
    if base_case
        return è
    else
        f(smaller_input)
}

The recursion calls should **STOP** to prevent infinite recursive regression!

The recursive method must define a **base case** that stops the recursive call.
Basic Illustrative Example

Factorial Function
Illustrative Example 1 - Factorial Function

- **Factorial Function: Formal Definition**
  - for any integer \( n \geq 0, \)

\[
\begin{array}{cccccccc}
0 & 1 & 1 & (1) & (2) & 3 & 2 & 1 & 0 \\
\end{array}
\]

Example

\[
5! = 5 \times 4 \times 3 \times 2 \times 1 = 120
\]

\[
= 5 \times (4 \times 3 \times 2 \times 1) = 5 \times 4!
\]

- **Recursive Definition**
  - for any integer \( n \geq 0, \)

\[
\begin{array}{cccccccc}
0 & 1 & 1 & (1)! & 0 \\
\end{array}
\]

The **factorial function** has a natural recursive definition.
Illustrative Example 1: Factorial Function

- Recursive Definition
  - for any integer \( n \geq 0 \),

\[
\begin{align*}
\text{Base case} & : \\
1! & = 1 \\
0! & = 0 \\
(1)! & = 1 \\
\end{align*}
\]

Recursive implementation of Factorial

```java
public static int factorial(int n){
    if (n == 0)
        return 1; // base case
    return n * factorial(n-1); // recursive case
}
```

```java
public static int recursiveFactorial(int n) throws IllegalArgumentException{
    if (n < 0)
        throw new IllegalArgumentException(); // argument must be positive
    else
        return factorial(n);
}
```
Illustrative Example 1: **Factorial Function**

Recursive implementation of Factorial

```java
public static int factorial(int n) {
    if (n == 0) {
        return 1; // base case
    } else {
        return n * factorial(n - 1); // recursive case
    }
}
```

- Each time the method executes:
  - it reduces the current problem to a smaller instance of the same problem
  - calls itself to solve the smaller problem
- The recursion stops when the base case is reached

Recursion Trace for n = 4

- 4! = 24
- return 4 * 6 = 24
- return 3 * 2 = 6
- return 2 * 1 = 2
- return 1 * 1 = 1
- return 1
- return 1 * 1 = 1
- return 2 * 1 = 2
- return 3 * 2 = 6
- return 4 * 6 = 24
- return 4! = 24

Illustrative Example 1: **Factorial Function**
Illustrative Example 1: Factorial Function

- **Recursive implementation of Factorial**

```java
public static int recursiveFactorial(int n) throws IllegalArgumentException{
    if (n < 0)
        throw new IllegalArgumentException(); // argument must be positive
    else
        return factorial(n);
}
```

```java
public static int factorial(int n){
    if (n == 0)
        return 1; // base case
    return n * factorial(n-1); // recursive case
}
```

- **Clearer (easier to understand)**

- **Running time: O(n)**

- **Iterative implementation of Factorial**

```java
public static int iterativeFactorial(int n) throws IllegalArgumentException{
    if (n < 0)
        throw new IllegalArgumentException(); // argument must be positive
    else{
        int fact = n;
        for(int i=n-1; i>1; i--)
            fact = fact * i;
        return fact;
    }
}
```

- **Running time: O(n)**
How does the computer implements a program?

A computer allocates **two areas of memory** for a program

- the **stack** to store information about method calls
  - When a piece of code calls a method, information about the call is placed on the stack.
  - When the method returns, that information is popped off the stack
- the **heap** to create and store objects, and perform calculations

The stack’s space size is much smaller than the heap’s one
Illustrative Example 1: Factorial Function

- Java uses a stack called "run-time" stack to keep track of the function calls.
- Each time a method is called recursively, an "activation record" is created and stored in the stack.
- When recursion returns, the corresponding activation is popped off the stack.
- Iteration simply 'jumps back' to the beginning of the loop.

A function call is usually more expensive than a jump.
Important Rule

- Do not use recursion as a substitute for a simple loop
  - Recursion could be very slow because of the extra memory and time overhead due to function/method calls!
Designing Recursive Algorithms
Recursion Method Design Rules

- Recursive methods **must terminate**
  - A recursive method **must** have at least one **base case**
  - A base case does not execute a recursive call. It **stops** the recursion

- Recursive methods should include
  - One or more **base cases**: refer to fixed values of the method
  - One or more **recursive cases**: definition of the function in terms of itself
    - Each recursive case **must** be a **smaller** version of itself
    - Each recursive case **must make progress** toward the base case (i.e. the base case should be reached)
Recursive Thinking

Is the problem “simple enough” to solve it directly?

Decompose: break the problem into one or more smaller subproblems $SP_1, SP_2, \ldots, SP_N$
Recursive Thinking ...

1. **Decompose** the problem into one or more smaller sub-problems.

   - Is the (sub-)problem “simple enough” to solve it directly?
     - Yes! Base case reached!

2. **Solve** each subproblem recursively:
   - solve(SP<sub>N</sub>), solve(SP<sub>N-1</sub>), ..., solve(SP<sub>2</sub>), solve(SP<sub>1</sub>)

3. **Combine results** into a solution that solves the original Problem.
Recursive Thinking ...

General Approach:

if the problem is "simple enough"
   solve it directly
else
   decompose into one or more smaller subproblems
   solve each subproblem recursively
   combine results into a solution to solve the original problem
General Recursive Design Strategy

1. **Identify** the **base case(s)** (for direct solution)
2. **Decompose:** define **subproblems** that have the same general structure as the original problem
   - Subproblems **must** be **smaller**
   - Subproblems **must work towards a base case**
3. **Combine** solutions from calls to other methods to obtain a solution for the original problem
Recursion - Keep in Mind

- A recursive method is a method that **invokes itself** as a subroutine with a **smaller input**, until a **base case** is reached.
- Base case not reached → infinite recursive regression → **Stack Overflow Error**
- Recursive programs not only simplify the algorithm design but also tend to give a cleaner code.
- Recursion is a powerful programming tool that in many cases (**not all cases**) can yield both **short** and **efficient** algorithms.
Recursion - Keep in Mind

- Recursion is not always the best solution
  - recursion might cause poor performance
  - Sometimes algorithms that are naturally expressed recursively must be rewritten without recursion
- Avoid using recursion in performance situations
  - Recursive calls take time and consume additional memory
Linear Search

- **Search: Problem Statement**
  - Given a target value X, return the index of X in the array, if such X exists.
  - Otherwise, return (-1).

- **Sequential Search** or **linear search** is a search technique used to find a target value (or key value) within a list (here an array).
  - It checks each element of the array for the target value from the first position to the last position in a linear progression (i.e. until a match is found **successful search**) or until all the elements have been searched **unsuccessful search**).
Linear Search

Unsuccessful search: search(45) → NOT_FOUND (-1)
Linear Search

<table>
<thead>
<tr>
<th></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><strong>data</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Successful search:</strong></td>
<td>search(12)</td>
<td></td>
<td></td>
<td></td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>17</td>
<td>5</td>
<td>90</td>
<td>12</td>
<td>44</td>
<td>38</td>
<td>84</td>
<td>77</td>
<td></td>
</tr>
</tbody>
</table>
Binary Search

Problem Description

Locate a target value within a sequence of n elements sorted in an array

Input: array data of n integer numbers sorted in ascending order
Target: the search target element
Output: index of the specified target in the given array of present, -1 otherwise

<table>
<thead>
<tr>
<th>data</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>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>-2</td>
<td>3</td>
<td>5</td>
<td>10</td>
<td>11</td>
<td>18</td>
<td>21</td>
<td>33</td>
<td>35</td>
<td>40</td>
<td>42</td>
<td>51</td>
<td>67</td>
<td>69</td>
<td>84</td>
</tr>
</tbody>
</table>
Binary Search …

- **Linear Search** (or Sequential Search) for **unsorted sequence**
  - Search Approach
    - Use a **loop** to **examine every element** until either **finding the target** or **exhausting** the data set
  - Running time: $O(n)$, where $n$ is the size of the array (data set)

- **Binary Search** – Precondition: **sorted** and **indexable** sequence of numbers

**Target:** 40

Arbitrary element $e = 11 < 40$

<table>
<thead>
<tr>
<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>
</tr>
</thead>
<tbody>
<tr>
<td>-2</td>
<td>3</td>
<td>5</td>
<td>10</td>
<td><strong>11</strong></td>
<td>18</td>
<td>21</td>
<td>33</td>
<td>35</td>
<td>40</td>
<td>42</td>
<td>51</td>
<td>67</td>
<td>69</td>
<td>84</td>
</tr>
</tbody>
</table>

The Target should be in this portion of the array if present

**Binary Search:** compare the target value to the median
**Binary Search**

**Binary Search** – Precondition: **sorted** and **indexable** sequence of numbers

- The algorithm maintains two parameters: low and high
- Initially, low = 0 and high = n-1

**Binary Search Design Strategy:**

Compare the target value to the **median** one, that is the element with index mid = (low + high)/2

1. If the target **equals** the median value (item at index mid),
   then **return mid** and the search terminates successfully

2. If the target is **less** than the median value,
   then **recur on the left half** of the sequence; (i.e. on the interval of indices [low, mid-1])

3. If the target is **greater** than the median value,
   then **recur on the right half** of the sequence; (i.e. the interval of indices [mid +1, high])

4. If **low > high** (i.e. the interval [low, high] is **empty**),
   then **return -1** and the search terminates unsuccessfully
**Binary Search** …

- **Binary Search** – Precondition: **sorted** and **indexable** sequence of numbers

**Target:** 40

If present, \( data[\text{low}] \leq \text{Target} \leq data[\text{high}] \)

**low** = (high + low)/2

Initially, low = 0 and high = n-1 = 14
Binary Search ...

- **Binary Search** – Precondition: sorted and indexable sequence of numbers

<table>
<thead>
<tr>
<th>low</th>
<th>mid</th>
<th>high</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4 5 6 7</td>
<td>8 9 10 11 12 13 14</td>
<td></td>
</tr>
</tbody>
</table>

**Target:** 40

<table>
<thead>
<tr>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>-2</td>
</tr>
</tbody>
</table>

- data[mid] = 33 < 40

<table>
<thead>
<tr>
<th>low</th>
<th>mid</th>
<th>high</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4 5 6 7</td>
<td>8 9 10 11 12 13 14</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>-2</td>
</tr>
</tbody>
</table>

- data[mid] = 51 > 40

<table>
<thead>
<tr>
<th>low</th>
<th>mid</th>
<th>high</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4 5 6 7</td>
<td>8 9 10 11 12 13 14</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>-2</td>
</tr>
</tbody>
</table>

- data[mid] == 40 (Target)

return 9
Binary Search – Keep in Mind

- **Precondition:** Sorted array
  - If the values in the array are arranged in ascending or descending order, then we call the array sorted.
- **Assumption:** The array is sorted in an ascendant order
- **Basic idea:** The search is performed by examining the median of a sorted array (the middle element)
  - If match, target element found.
  - Otherwise, if target element is smaller than the median, search in the subarray that is to the left of the median.
  - Otherwise, search in the subarray that is to the right of the median.

This procedure presumes that the subarray is not empty; if it is, the target element is not found.
Binary Search - Example

Search for 77 in the array “data”

Execution Trace (trace of visited nodes):
38, 77

Running Time: $O(\log(n))$
Where the problem size $n$ is the number of items stored in the array