PRACTICAL ASSIGNMENT - MARKING REPORT

1. PERSONAL DATA

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Group number : 50** | | | | |
| No | Name | ID | Programme | Total Marks |
| 1. | Koh Khai Jeck | 2304740 | SE |  |
| 2. | Leon Siow Yi Hong | 2204403 | SE |  |
| 3. | Quak Jing | 2205378 | SE |  |
| 4. |  |  |  |  |

1. SUBMISSION STATUS

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| No soft copy/ Upload wrong file(s) | Late submission of softcopy | No hardcopy | Late submission of hardcopy | No issue |
|  |  |  |  |  |

1. COMPILATION AND RUNNING

|  |  |  |
| --- | --- | --- |
| Does not compile/Bytecode & batch file do not work | Compile but no output/ wrong output/ run-time error | Compile and produce output |
|  |  |  |

1. PRESENTATION OF SOURCE CODES(3%)
2. Indent Style (1.5%) Poor Inconsistent Good
3. Identifier names (1.5%) Poor choice Meaningful Meaningful and good naming convention
4. PROGRAM COMPONENT (57% + 3%)

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Program Components | Missing/ Does not work | Major errors | Minor errors | Not robust | No issue/ Excellent design | Max marks | Marks obtained |
| Framework Design (Use of interfaces and abstract classes) |  |  |  |  |  | 10 |  |
| Classes for storing objects (data structures/containers) |  |  |  |  |  | 12 |  |
| Bin Packing Algorithms (at least 2) |  |  |  |  |  | 16 |  |
| Test program (main program, set of bins and set of objects) |  |  |  |  |  | 14 |  |
| Exception and error handling |  |  |  |  |  | 5 |  |
| Presentation of source codes |  |  |  |  |  | 3 |  |
|  |  |  |  |  | Total | 60 |  |

1. REPORT AND OTHER COMPONENT (40%)

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Components | Missing | Poor | Average | Good | Excellent | Max marks | Marks obtained |
| The proposed solution and design (data structures and algorithms) |  |  |  |  |  | 8 |  |
| Discussion (efficiency and complexities) |  |  |  |  |  | 12 |  |
| Flowchart |  |  |  |  |  | 5 |  |
| UML Diagram |  |  |  |  |  | 5 |  |
| Sample input and test cases |  |  |  |  |  | 5 |  |
| Screenshots |  |  |  |  |  | 5 |  |
|  |  |  |  |  | Total | 40 |  |

Table of Contents

[1.0 Proposed Solution 2](#_Toc197119227)

[1.1 People (Items) 2](#_Toc197119228)

[1.2 Elevator (Container) 2](#_Toc197119229)

[1.3 Bin Packing Algorithms (Strategy) 3](#_Toc197119230)

[2.0 Discussion and Complexity Analysis 4](#_Toc197119231)

[2.1 Efficiency of Algorithms 4](#_Toc197119232)

[2.2 Complexity of Algorithms 5](#_Toc197119233)

[3.0 Flowchart of Bin Packing Algorithms 7](#_Toc197119234)

[3.1 First Fit Algorithm 7](#_Toc197119235)

[3.2 Best Fit Algorithm 8](#_Toc197119236)

[3.3 Next Fit Algorithm 9](#_Toc197119237)

[3.4 Worst Fit Algorithm 10](#_Toc197119238)

[4.0 UML Diagram of Java Program 11](#_Toc197119239)

[5.0 Sample of Input Data and Test Cases 12](#_Toc197119240)

[5.1 Normal Case 1 (Selecting One Algorithm) 12](#_Toc197119241)

[5.2 Normal Case 2 (Selecting All Algorithms) 12](#_Toc197119242)

[5.3 Edge Case 1 (Elevator Specification Too Small) 13](#_Toc197119243)

[6.0 Sample Output(s) 14](#_Toc197119244)

[6.1 Normal Case 1 (Selecting One Algorithm) 14](#_Toc197119245)

[6.2 Normal Case 2 (Selecting All Algorithms) 14](#_Toc197119246)

[6.3 Edge Case 1 (Elevator Specification Too Small) 15](#_Toc197119247)

[7.0 Print Out of Java Program 16](#_Toc197119248)

[7.1 ElevatorList.java 16](#_Toc197119249)

[7.2 ElevatorAbstractList.java 18](#_Toc197119250)

[7.3 ElevatorArrayList.java 19](#_Toc197119251)

[7.4 QueueLinkedList.java 23](#_Toc197119252)

[7.5 ElevatorBin.java 30](#_Toc197119253)

[7.6 ElevatorQueue.java 33](#_Toc197119254)

[7.7 Person.java 35](#_Toc197119255)

[7.8 Algorithms.java 36](#_Toc197119256)

[7.9 AlgorithmTest.java (Main Program) 42](#_Toc197119257)

[7.10 PersonGenerator (Bonus Program) 46](#_Toc197119258)

# 1.0 Proposed Solution

In our daily life, clearing a queue that is using the elevator is a common headache due to unoptimized packing of people in the elevator. For this bin-packing assignment, the main goal is to minimize the number of elevators needed to compensate a queue of people, where people will be optimally packed into elevator. In this case, the people are the items, and the elevator is the container. The solution to achieve said goal are strategies that implements bin-packing algorithms.

## 1.1 People (Items)

A person class declared in “Person.java” outlines the attribute of a person. The attributes of a person are as follows:

|  |  |
| --- | --- |
| **Attribute** | **Representation** |
| area (m2) | The area occupied when standing in an elevator |
| weight (kg) | The load that the person will add on the elevator |

A group of persons is identified as people, where the queue data structure declared in “ElevatorQueue.java” is used to store each individual person. The reason for picking queue as the suitable data is to simulate real world scenario where people queue up to enter the elevators on a first come first serve basis.

## 1.2 Elevator (Container)

An elevator class declared in “ElevatorBin.java” outlines the attribute of an elevator. The attributes of an elevator are as follows:

|  |  |
| --- | --- |
| **Attribute** | **Representation** |
| bin | The people that are currently in the elevator |
| fullLoad (kg) | The maximum load of people that the elevator can handle |
| fullArea (m2) | The dimension of the elevator’s floor to stand on |
| currentLoad (kg) | The current load of people in the elevator |
| currentArea (m2) | The current area occupied by people in the elevator |

The elevator class is designed like a stack to capture the nature of first in last out when people are entering and exiting the elevator. A group of elevator is identified as elevators, where the array list data structure declared in “ElevatorArrayList.java” is used to store the elevator used. The reason for picking array list as the suitable data structure is because of the constant time indexing operation, which is commonly used in bin-packing algorithms.

## 1.3 Bin Packing Algorithms (Strategy)

To pack the people into the elevators efficiently, four popular approximation bin packing algorithms were used. The algorithms implemented are:

|  |  |
| --- | --- |
| **Algorithms** | **Strategy** |
| First Fit (FF) | People enter the first elevator that can handle them. |
| Best Fit (BF) | People enter the elevator with the least area available left. |
| Next Fit (NF) | People only enter the current elevator. |
| Worst Fit (WF) | People enter the elevator with the most area available left. |

These algorithms serve as an experimental societal norm of how people should approach entering elevator enter when queuing. Multiple algorithms are implemented to better showcase the difference in efficiency and complexity.

# 2.0 Discussion and Complexity Analysis

## 2.1 Efficiency of Algorithms

Each of the algorithms are tested with the same queue of people. The test is performed three separate times with different specifications of elevator. The documented results are as follows:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **fullLoad (kg)** | **fullArea (m2)** | **Algorithms** | **First Fit** | **Best Fit** | **Next Fit** | **Worst Fit** |
| 800 | 2 | **Number of Elevators Used** | 10863 | 10865 | 11305 | 10879 |
| 1700 | 3.75 | 5564 | 5564 | 5639 | 5562 |
| 1925 | 5 | 4302 | 4303 | 4380 | 4312 |

As the full load and area of the elevator decreases, the difference in performance is increasingly obvious. As observed from the results, first fit and best fit performed the best with very little difference in performance. The best performing algorithm is followed by worst fit and then next fit.

From this observation, people should either enter the first elevator that can handle them or enter the elevator with the least area available left to minimize the elevators needed to clear the queue of people. The people can consider entering the elevator with the most area available left if they prefer open spaces as it is on average only 0.11% less inefficient than first fit and best fit. On the contrary, people should avoid only entering the current elevator as it is on average 2.3% less efficient than the other algorithms.

## 2.2 Complexity of Algorithms

By studying the algorithm, the time complexity of each algorithm is identified. The identified time complexity are as follows:

|  |  |  |
| --- | --- | --- |
| **Algorithms** | **Time Complexity** | **Iteration/Loop Analysis** |
| First Fit | O(n2) | Outer Loop: Iterate through each person in people.  Inner Loop (layer 1): Iterate through each elevator in elevators to find the first elevator that can handle the person. |
| Best Fit | O(n2) | Outer Loop: Iterate through each person in people.  Inner Loop 1 (layer 1): Iterate through each elevator in elevators to find all the elevator that can handle the person  Inner Loop 2 (layer 1): Iterate through all the elevator that can handle the person to find the least space remaining |
| Next Fit | O(n) | Outer Loop: Iterate through each person in people. |
| Worst Fit | O(n2) | Outer Loop: Iterate through each person in people.  Inner Loop 1 (layer 1): Iterate through each elevator in elevators to find all the elevator that can handle the person  Inner Loop 2 (layer 1): Iterate through all the elevator that can handle the person to find the most space remaining |

While testing the efficiency of the algorithms, the runtime (ms) of each algorithm was also captured. The documented results are as follows:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Load (kg)** | **Area (m2)** | **Algorithms** | **First Fit** | **Best Fit** | **Next Fit** | **Worst Fit** |
| 800 | 2 | **Runtime (ms)** | 896 | 2464 | 5 | 2533 |
| 1700 | 3.75 | 288 | 884 | 0 | 1050 |
| 1925 | 5 | 245 | 742 | 0 | 956 |

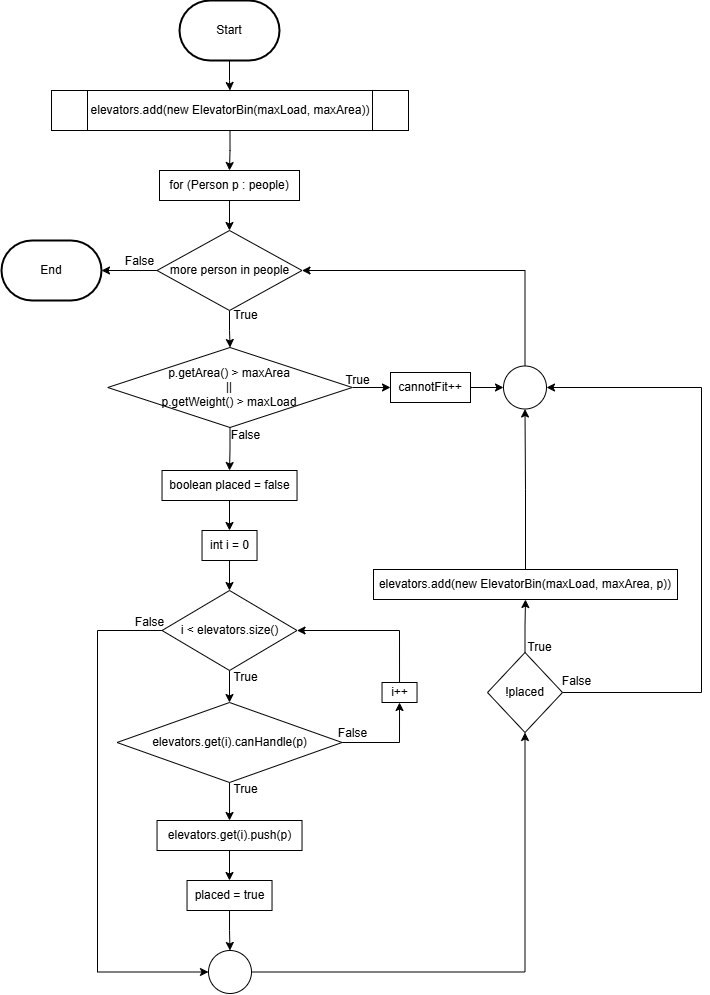
As the full load and area of the elevator decreases, the difference in runtime is increasingly obvious. As observed from the results, next fit is the fastest algorithm by a huge margin, followed by first fit. The two of the slowest algorithms are best fit and worst fit with very little difference in runtime.

From the documentation, the complexity of each algorithm can be explained by the number and nesting of loops. Next fit is the fastest as it only iterates through the queue of people once and immediately pack them into the elevator, resulting in linear time. First fit is slower than next fit as its algorithm iterates through the queue of people and has a loop nested to iterate through the elevators to find the first elevator that can handle the current person, resulting in quadratic time.

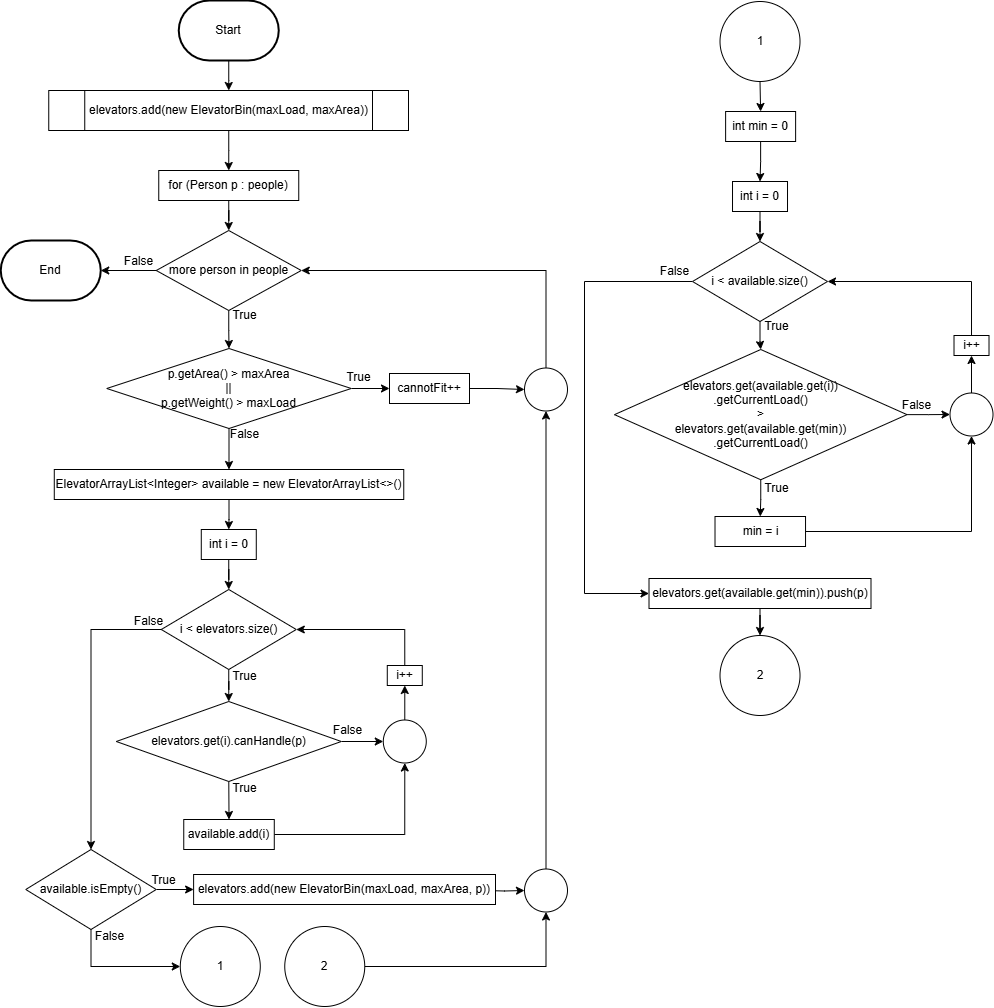
Best fit and worst fit is the slowest because their algorithm iterates through the queue of people and has 2 loops nested to iterate through the elevators to find all the elevators that can handle the current person and to find the elevator with the least/most space remaining, resulting in quadratic time. Furthermore, the longer runtime is due to the 2 loops in the first nested layer, in contrast to first fit which only have 1 loop in the first nested layer.

# 3.0 Flowchart of Bin Packing Algorithms

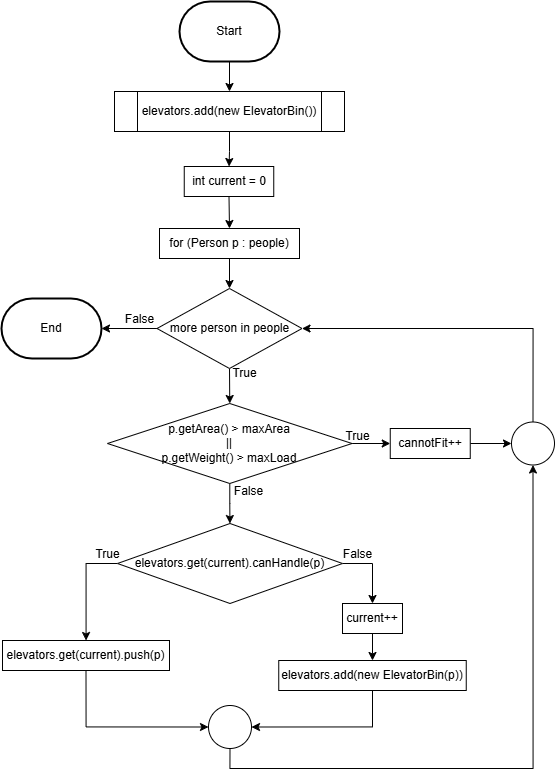
## 3.1 First Fit Algorithm



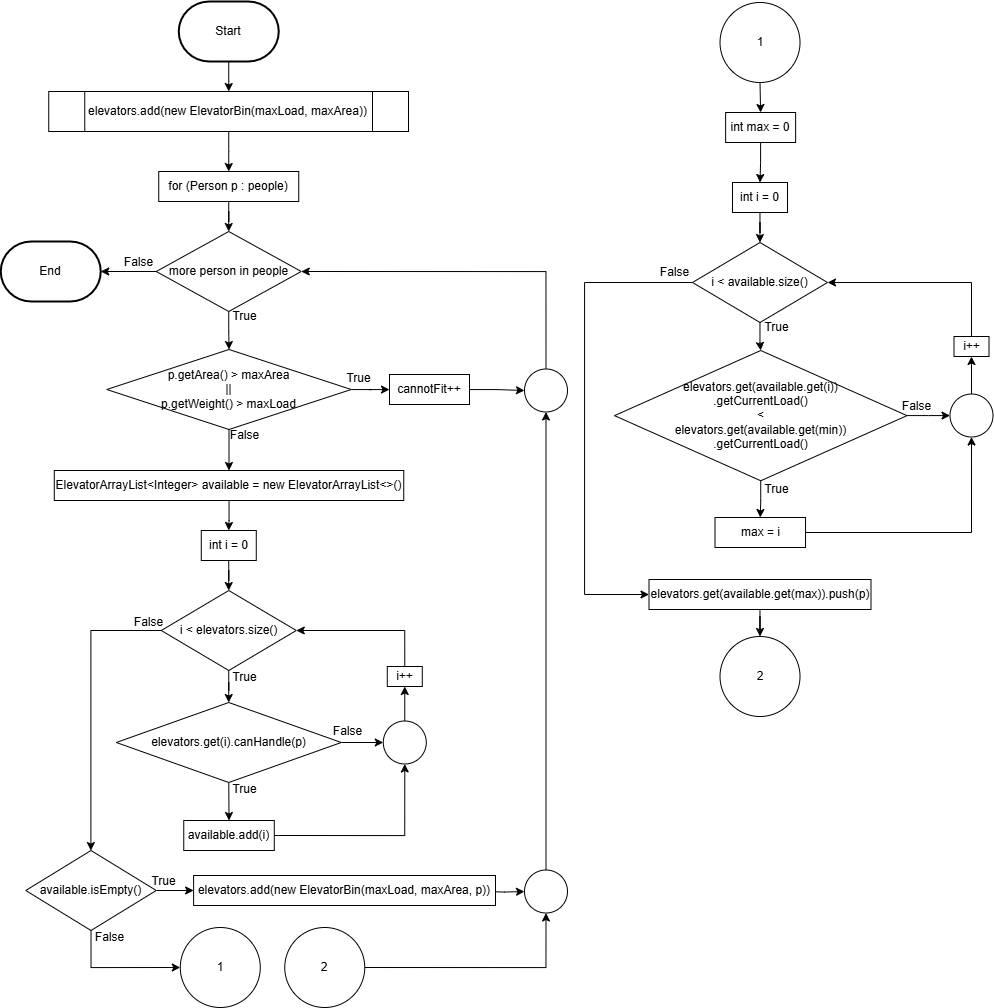
## 3.2 Best Fit Algorithm



## 3.3 Next Fit Algorithm



## 3.4 Worst Fit Algorithm



# 4.0 UML Diagram of Java Program

# 5.0 Sample of Input Data and Test Cases

## 5.1 Normal Case 1 (Selecting One Algorithm)

A screenshot of a computer program

AI-generated content may be incorrect.

A black background with white text

AI-generated content may be incorrect.

## 5.2 Normal Case 2 (Selecting All Algorithms)

A screenshot of a computer program

AI-generated content may be incorrect.

## 5.3 Edge Case 1 (Elevator Specification Too Small)

A screenshot of a computer

AI-generated content may be incorrect.

# 6.0 Sample Output(s)

## 6.1 Normal Case 1 (Selecting One Algorithm)

A black background with white text

AI-generated content may be incorrect.

A computer screen with white text

AI-generated content may be incorrect.

## 6.2 Normal Case 2 (Selecting All Algorithms)

A screenshot of a computer program

AI-generated content may be incorrect.

## 6.3 Edge Case 1 (Elevator Specification Too Small)

A black screen with white text

AI-generated content may be incorrect.

# 7.0 Print Out of Java Program

## 7.1 ElevatorList.java

public interface ElevatorList<E> extends java.lang.Iterable<E> {

    /\*\* Add a new element at the end of this list \*/

    public void add(E e);

    /\*\* Add a new element at the specified index in this list \*/

    public void add(int index, E e);

    /\*\* Clear the list \*/

    public void clear();

    /\*\* Return true if this list contains the element \*/

    public boolean contains(E e);

    /\*\* Return the element from this list at the specified index \*/

    public E get(int index);

    /\*\*

     \* Return the index of the first matching element in this list.

     \* Return -1 if no match.

     \*/

    public int indexOf(E e);

    /\*\* Return true if this list contains no elements \*/

    public boolean isEmpty();

    /\*\*

     \* Return the index of the last matching element in this list

     \* Return -1 if no match.

     \*/

    public int lastIndexOf(E e);

    /\*\*

     \* Remove the first occurrence of the element o from this list.

     \* Shift any subsequent elements to the left.

     \* Return true if the element is removed.

     \*/

    public boolean remove(E e);

    /\*\*

     \* Remove the element at the specified position in this list

     \* Shift any subsequent elements to the left.

     \* Return the element that was removed from the list.

     \*/

    public E remove(int index);

    /\*\*

     \* Replace the element at the specified position in this list

     \* with the specified element and returns the new set.

     \*/

    public Object set(int index, E e);

    /\*\* Return the number of elements in this list \*/

    public int size();

    /\*\* Load sample elements from a file into the List \*/

    public void load(String fileName);

}

## 7.2 ElevatorAbstractList.java

public abstract class ElevatorAbstractList<E> implements ElevatorList<E> {

  protected int size = 0; // The size of the list

  /\*\* Create a default list \*/

  protected ElevatorAbstractList() {

  }

  /\*\* Create a list from an array of objects \*/

  protected ElevatorAbstractList(E[] objects) {

    for (int i = 0; i < objects.length; i++)

      add(objects[i]);

  }

  @Override /\*\* Add a new element at the end of this list \*/

  public void add(E e) {

    add(size, e);

  }

  @Override /\*\* Return true if this list contains no elements \*/

  public boolean isEmpty() {

    return size == 0;

  }

  @Override /\*\* Return the number of elements in this list \*/

  public int size() {

    return size;

  }

  @Override /\*\*

             \* Remove the first occurrence of the element e

             \* from this list. Shift any subsequent elements to the left.

             \* Return true if the element is removed.

             \*/

  public boolean remove(E e) {

    if (indexOf(e) >= 0) {

      remove(indexOf(e));

      return true;

    } else

      return false;

  }

}

## 7.3 ElevatorArrayList.java

import java.io.File;

import java.io.FileNotFoundException;

import java.util.Scanner;

public class ElevatorArrayList<E> extends ElevatorAbstractList<E> {

    public static final int INITIAL\_CAPACITY = 16;

    private E[] data = (E[]) new Object[INITIAL\_CAPACITY];

    /\*\* Create a default list \*/

    public ElevatorArrayList() {

    }

    /\*\* Create a list from an array of objects \*/

    public ElevatorArrayList(E[] objects) {

        for (int i = 0; i < objects.length; i++)

            add(objects[i]); // Warning: don�t use super(objects)!

    }

    @Override /\*\* Add a new element at the specified index \*/

    public void add(int index, E e) {

        ensureCapacity();

        // Move the elements to the right after the specified index

        for (int i = size - 1; i >= index; i--)

            data[i + 1] = data[i];

        // Insert new element to data[index]

        data[index] = e;

        // Increase size by 1

        size++;

    }

    /\*\* Create a new larger array, double the current size + 1 \*/

    private void ensureCapacity() {

        if (size >= data.length) {

            E[] newData = (E[]) (new Object[size \* 2 + 1]);

            System.arraycopy(data, 0, newData, 0, size);

            data = newData;

        }

    }

    @Override /\*\* Clear the list \*/

    public void clear() {

        data = (E[]) new Object[INITIAL\_CAPACITY];

        size = 0;

    }

    @Override /\*\* Return true if this list contains the element \*/

    public boolean contains(E e) {

        for (int i = 0; i < size; i++)

            if (e.equals(data[i]))

                return true;

        return false;

    }

    @Override /\*\* Return the element at the specified index \*/

    public E get(int index) {

        checkIndex(index);

        return data[index];

    }

    private void checkIndex(int index) {

        if (index < 0 || index >= size)

            throw new IndexOutOfBoundsException("Index: " + index + ", Size: " + size);

    }

    @Override /\*\*

               \* Return the index of the first matching element

               \* in this list. Return -1 if no match.

               \*/

    public int indexOf(E e) {

        for (int i = 0; i < size; i++)

            if (e.equals(data[i]))

                return i;

        return -1;

    }

    @Override /\*\*

               \* Return the index of the last matching element

               \* in this list. Return -1 if no match.

               \*/

    public int lastIndexOf(E e) {

        for (int i = size - 1; i >= 0; i--)

            if (e.equals(data[i]))

                return i;

        return -1;

    }

    @Override /\*\*

               \* Remove the element at the specified position

               \* in this list. Shift any subsequent elements to the left.

               \* Return the element that was removed from the list.

               \*/

    public E remove(int index) {

        checkIndex(index);

        E e = data[index];

        // Shift data to the left

        for (int j = index; j < size - 1; j++)

            data[j] = data[j + 1];

        data[size - 1] = null; // This element is now null

        // Decrement size

        size--;

        return e;

    }

    @Override /\*\*

               \* Replace the element at the specified position

               \* in this list with the specified element.

               \*/

    public E set(int index, E e) {

        checkIndex(index);

        E old = data[index];

        data[index] = e;

        return old;

    }

    @Override

    public String toString() {

        StringBuilder result = new StringBuilder("[");

        for (int i = 0; i < size; i++) {

            result.append(data[i]);

            if (i < size - 1)

                result.append(", ");

        }

        return result.toString() + "]";

    }

    /\*\* Trims the capacity to current size \*/

    public void trimToSize() {

        if (size != data.length) {

            E[] newData = (E[]) (new Object[size]);

            System.arraycopy(data, 0, newData, 0, size);

            data = newData;

        } // If size == capacity, no need to trim

    }

    @Override /\*\* Override iterator() defined in Iterable \*/

    public java.util.Iterator<E> iterator() {

        return new ArrayListIterator();

    }

    private class ArrayListIterator

            implements java.util.Iterator<E> {

        private int current = 0; // Current index

        @Override

        public boolean hasNext() {

            return (current < size);

        }

        @Override

        public E next() {

            return data[current++];

        }

        @Override

        public void remove() {

            ElevatorArrayList.this.remove(current);

        }

    }

    @Override // Loads data from a file into the LinkedList

    public void load(String fileName) {

        try {

            File file = new File(fileName);

            Scanner fileReader = new Scanner(file);

            while (fileReader.hasNextLine()) {

                double pArea = fileReader.nextFloat();

                double pWeight = fileReader.nextFloat();

                add((E) new Person(pArea, pWeight));

            }

            System.out.println("Data is loaded from " + fileName + " successfully.\n");

            fileReader.close();

        } catch (FileNotFoundException e) {

            System.out.println("Failed to find People.txt");

            e.printStackTrace();

        }

    }

}

## 7.4 QueueLinkedList.java

import java.io.File;

import java.io.FileNotFoundException;

import java.util.Scanner;

public class QueueLinkedList<E> extends ElevatorAbstractList<E> {

  private Node<E> head, tail;

  /\*\* Create a default list \*/

  public QueueLinkedList() {

  }

  /\*\* Create a list from an array of objects \*/

  public QueueLinkedList(E[] objects) {

    super(objects);

  }

  /\*\* Return the head element in the list \*/

  public E getFirst() {

    if (size == 0) {

      return null;

    } else {

      return head.element;

    }

  }

  /\*\* Return the last element in the list \*/

  public E getLast() {

    if (size == 0) {

      return null;

    } else {

      return tail.element;

    }

  }

  /\*\* Add an element to the beginning of the list \*/

  public void addFirst(E e) {

    Node<E> newNode = new Node<E>(e); // Create a new node

    newNode.next = head; // link the new node with the head

    head = newNode; // head points to the new node

    size++; // Increase list size

    if (tail == null) // the new node is the only node in list

      tail = head;

  }

  /\*\* Add an element to the end of the list \*/

  public void addLast(E e) {

    Node<E> newNode = new Node<E>(e); // Create a new for element e

    if (tail == null) {

      head = tail = newNode; // The new node is the only node in list

    } else {

      tail.next = newNode; // Link the new with the last node

      tail = tail.next; // tail now points to the last node

    }

    size++; // Increase size

  }

  @Override /\*\*

             \* Add a new element at the specified index

             \* in this list. The index of the head element is 0

             \*/

  public void add(int index, E e) {

    if (index == 0) {

      addFirst(e);

    } else if (index >= size) {

      addLast(e);

    } else {

      Node<E> current = head;

      for (int i = 1; i < index; i++) {

        current = current.next;

      }

      Node<E> temp = current.next;

      current.next = new Node<E>(e);

      (current.next).next = temp;

      size++;

    }

  }

  /\*\*

   \* Remove the head node and

   \* return the object that is contained in the removed node.

   \*/

  public E removeFirst() {

    if (size == 0) {

      return null;

    } else {

      Node<E> temp = head;

      head = head.next;

      size--;

      if (head == null) {

        tail = null;

      }

      return temp.element;

    }

  }

  /\*\*

   \* Remove the last node and

   \* return the object that is contained in the removed node.

   \*/

  public E removeLast() {

    if (size == 0) {

      return null;

    } else if (size == 1) {

      Node<E> temp = head;

      head = tail = null;

      size = 0;

      return temp.element;

    } else {

      Node<E> current = head;

      for (int i = 0; i < size - 2; i++) {

        current = current.next;

      }

      Node<E> temp = tail;

      tail = current;

      tail.next = null;

      size--;

      return temp.element;

    }

  }

  @Override /\*\*

             \* Remove the element at the specified position in this

             \* list. Return the element that was removed from the list.

             \*/

  public E remove(int index) {

    if (index < 0 || index >= size) {

      return null;

    } else if (index == 0) {

      return removeFirst();

    } else if (index == size - 1) {

      return removeLast();

    } else {

      Node<E> previous = head;

      for (int i = 1; i < index; i++) {

        previous = previous.next;

      }

      Node<E> current = previous.next;

      previous.next = current.next;

      size--;

      return current.element;

    }

  }

  @Override /\*\* Override toString() to return elements in the list \*/

  public String toString() {

    StringBuilder result = new StringBuilder("[");

    Node<E> current = head;

    for (int i = 0; i < size; i++) {

      result.append(current.element);

      current = current.next;

      if (current != null) {

        result.append(", "); // Separate two elements with a comma

      } else {

        result.append("]"); // Insert the closing ] in the string

      }

    }

    return result.toString();

  }

  @Override /\*\* Clear the list \*/

  public void clear() {

    size = 0;

    head = tail = null;

  }

  @Override /\*\* Return true if this list contains the element e \*/

  public boolean contains(E e) {

    Node<E> current = head;

    while (current != null) {

      if (current.element.equals(e)) {

        return true;

      }

      current = current.next;

    }

    return false;

  }

  @Override /\*\* Return the element at the specified index \*/

  public E get(int index) {

    checkIndex(index);

    Node<E> current = head;

    for (int i = 0; i < index; i++) {

      current = current.next;

    }

    return current.element;

  }

  @Override /\*\*

             \* Return the index of the head matching element in this list. Return -1 if no

             \* match.

             \*/

  public int indexOf(E e) {

    Node<E> current = head;

    int index = 0;

    while (current != null) {

      if (current.element.equals(e)) {

        return index;

      }

      current = current.next;

      index++;

    }

    return -1;

  }

  @Override /\*\*

             \* Return the index of the last matching element in this list. Return -1 if no

             \* match.

             \*/

  public int lastIndexOf(E e) {

    Node<E> current = head;

    int index = 0;

    int lastIndex = -1;

    while (current != null) {

      if (current.element.equals(e)) {

        lastIndex = index;

      }

      current = current.next;

      index++;

    }

    return lastIndex;

  }

  @Override /\*\*

             \* Replace the element at the specified position in this list with the specified

             \* element.

             \*/

  public E set(int index, E e) {

    checkIndex(index);

    Node<E> current = head;

    for (int i = 0; i < index; i++) {

      current = current.next;

    }

    E oldElement = current.element;

    current.element = e;

    return oldElement;

  }

  @Override /\*\* Override iterator() defined in Iterable \*/

  public java.util.Iterator<E> iterator() {

    return new LinkedListIterator();

  }

  private void checkIndex(int index) {

    if (index < 0 || index >= size)

      throw new IndexOutOfBoundsException("Index: " + index + ", Size: " + size);

  }

  private class LinkedListIterator

      implements java.util.Iterator<E> {

    private Node<E> current = head; // Current index

    @Override

    public boolean hasNext() {

      return (current != null);

    }

    @Override

    public E next() {

      E e = current.element;

      current = current.next;

      return e;

    }

    @Override

    public void remove() {

      if (current == null) {

        throw new IllegalStateException("Next method has not been called or the element has already been removed.");

      }

      Node<E> previous = null;

      Node<E> temp = head;

      // Traverse the list to find the node before the current node

      while (temp != null && temp != current) {

        previous = temp;

        temp = temp.next;

      }

      if (previous == null) { // Removing the head node

        head = head.next;

        if (head == null) {

          tail = null; // List is now empty

        }

      } else {

        previous.next = current.next;

        if (current == tail) {

          tail = previous; // Update tail if last node is removed

        }

      }

      size--;

      current = null;

    }

  }

  private static class Node<E> {

    E element;

    Node<E> next;

    public Node(E element) {

      this.element = element;

    }

  }

  @Override // Loads data from a file into the LinkedList

  public void load(String fileName) {

    try {

      File file = new File(fileName);

      Scanner fileReader = new Scanner(file);

      while (fileReader.hasNextLine()) {

        double pArea = fileReader.nextFloat();

        double pWeight = fileReader.nextFloat();

        add((E) new Person(pArea, pWeight));

      }

      System.out.println("Data is loaded from " + fileName + " successfully.\n");

      fileReader.close();

    } catch (FileNotFoundException e) {

      System.out.println("Failed to find People.txt");

      e.printStackTrace();

    }

  }

}

## 7.5 ElevatorBin.java

public class ElevatorBin {

    // Elevator is used as the bin for the bin packing problem

    private ElevatorArrayList<Person> bin = new ElevatorArrayList<>();

    private double fullLoad = 0;

    private double fullArea = 0;

    private double currentLoad = 0;

    private double currentArea = 0;

    // Create a default elevator

    public ElevatorBin() {

        // Default values of the elevator

        fullLoad = 800;

        fullArea = 2.25;

    }

    public ElevatorBin(double load, double area) {

        // Create a elevator with the given load and area

        fullLoad = load;

        fullArea = area;

    }

    // Create a elevator with person, p inside

    public ElevatorBin(double load, double area, Person p) {

        fullLoad = load;

        fullArea = area;

        push(p);

    }

    // return the bin of the elevator

    public ElevatorArrayList<Person> getBin() {

        return bin;

    }

    // return number of people in the elevator

    public int getSize() {

        return bin.size();

    }

    // return the current load of people in the elevator

    public double getCurrentLoad() {

        return currentLoad;

    }

    // return the current occupied area of the elevator

    public double getCurrentArea() {

        return currentArea;

    }

    // return the last person to enter the elevator

    public Person peek() {

        return bin.get(getSize() - 1);

    }

    // add a new person to enter the elevator

    public void push(Person p) {

        if ((currentLoad + p.getWeight()) > fullLoad) {

            System.out.println("Elevator Load Limit Exceeded");

            return;

        }

        if ((currentArea + p.getArea()) > fullArea) {

            System.out.println("Elevator Area Limit Exceeded");

            return;

        }

        bin.add(p);

        currentLoad += p.getWeight();

        currentArea += p.getArea();

    }

    // return and remove the last person to enter the elevator

    public Person pop() {

        try {

            Person person = bin.get(getSize() - 1);

            bin.remove(getSize() - 1);

            currentLoad -= person.getWeight();

            currentArea -= person.getArea();

            return person;

        } catch (Exception e) {

            System.out.println("The Elevator is Empty");

            return null;

        }

    }

    // checking if the elevator is empty

    public boolean isEmpty() {

        return bin.isEmpty();

    }

    // checking if the elevator can accept the person

    public boolean canHandle(Person p) {

        return ((currentArea + p.getArea()) < fullArea) && ((currentLoad + p.getWeight()) < fullLoad);

    }

    // printing out the person in the elevator

    @Override

    public String toString() {

        return "Elevator: " + bin.toString();

    }

}

## 7.6 ElevatorQueue.java

public class ElevatorQueue implements Iterable<Person> {

    private QueueLinkedList<Person> queue = new QueueLinkedList<>();

    // Create a default elevator queue

    public ElevatorQueue() {

    }

    // return the queue of the elevator

    public QueueLinkedList<Person> getQueue() {

        return queue;

    }

    // add a person to the queue

    public void enqueue(Person p) {

        queue.addLast(p);

    }

    // remove and return the first person in the queue

    public Person dequeue() {

        if (isEmpty()) {

            System.out.println("Queue is empty");

            return null;

        }

        Person person = queue.getFirst();

        queue.removeFirst();

        return person;

    }

    // return number of people in the elevator queue

    public int getSize() {

        return queue.size();

    }

    // return true if the elevator queue is empty

    public boolean isEmpty() {

        return queue.size() == 0;

    }

    @Override

    public String toString() {

        return "Queue: " + queue.toString();

    }

    @Override

    public java.util.Iterator<Person> iterator() {

        return queue.iterator();

    }

    // Loads data from a file into the queue

    public void load(String fileName) {

        queue.load(fileName);

    }

}

## 7.7 Person.java

import java.lang.Comparable;

public class Person implements Comparable<Person> {

    // Person is the objects to be packed into the bin(elevator)

    private double area;

    private double weight;

    // Create a default person

    public Person() {

        area = 0;

        weight = 0;

    }

    // Create a person with input area and weight

    public Person(double area, double weight) {

        this.area = area;

        this.weight = weight;

    }

    // return the area occupied by the person

    public double getArea() {

        return area;

    }

    // return the weight of the person

    public double getWeight() {

        return weight;

    }

    // comparison between two person

    @Override

    public int compareTo(Person P) {

        return (int) this.area - (int) P.area;

    }

}

## 7.8 Algorithms.java

import java.text.DecimalFormat;

public class Algorithms {

    private String algoName;

    // All the elevator used will be stored in the elevators

    private ElevatorArrayList<ElevatorBin> elevators = new ElevatorArrayList<>();

    // Number of people that cannot fit into any elevator

    private int cannotFit = 0;

    // Specification of the elevators

    double maxLoad = 0;

    double maxArea = 0;

    // Number format of the result

    DecimalFormat numberFormat = new DecimalFormat("#0.00");

    // Constructor for specified algorithm

    public Algorithms(String algo, double maxLoad, double maxArea, ElevatorQueue people) {

        algoName = algo;

        this.maxLoad = maxLoad;

        this.maxArea = maxArea;

        switch (algo) {

            case "First Fit":

                firstFit(people);

                break;

            case "Best Fit":

                bestFit(people);

                break;

            case "Next Fit":

                nextFit(people);

                break;

            case "Worst Fit":

                worstFit(people);

                break;

            default:

                System.out.println("Invalid algorithm name: " + algo);

                System.out.println("Please use one of the following algorithms:");

                System.out.println("Next Fit, Best Fit, Worst Fit, First Fit");

                break;

        }

    }

    // Print out the number of elevators used by the algorithm

    public void getResult() {

        System.out.println(algoName);

        System.out.println("Elevators needed = " + elevators.size());

        if (cannotFit > 0) {

            System.out.println("Number of people that cannot fit into any elevator = " + cannotFit);

        }

    }

    // Print out the statistic of the algorithm

    public void getBins() {

        // Calculate area wasted by each elevator

        Double wastedArea = elevators.size() \* 2.25;

        for (ElevatorBin elevator : elevators) {

            wastedArea -= elevator.getCurrentArea();

        }

        System.out.println("Area wasted (m^2) = " + numberFormat.format(wastedArea));

        // Calculate load wasted by each elevator

        Double wastedLoad = elevators.size() \* 800.0;

        for (ElevatorBin elevator : elevators) {

            wastedLoad -= elevator.getCurrentLoad();

        }

        System.out.println("Load wasted (kg)  = " + numberFormat.format(wastedLoad));

        System.out.println("");

        // Print out the detail of each elevator in elevators

        int count = 1;

        for (ElevatorBin elevator : elevators) {

            System.out.println("Elevator " + count + " (Number of People= " + elevator.getSize() + ", Area Occupied= "

                    + numberFormat.format(elevator.getCurrentArea()) + ", Total Load= " +

                    numberFormat.format(elevator.getCurrentLoad()) + ")");

            System.out.println("Detail of Each Person");

            System.out.println("=====================");

            System.out.println("Area  Weight");

            System.out.println("---------------------");

            ElevatorArrayList<Person> bin = elevator.getBin();

            for (Person p : bin) {

                System.out.println(numberFormat.format(p.getArea()) + "  " + numberFormat.format(p.getWeight()));

            }

            System.out.println("---------------------");

            System.out.println("");

            count++;

        }

    }

    // First Fit Algorithm execution

    public void firstFit(ElevatorQueue people) {

        // Add the first elevator into elevators

        elevators.add(new ElevatorBin(maxLoad, maxArea));

        for (Person p : people) {

            // If the person is too big for the elevator, increment cannotFit and continue

            if (p.getArea() > maxArea || p.getWeight() > maxLoad) {

                cannotFit++;

                continue;

            }

            boolean placed = false;

            // Try placing the person into the first elevator that can handle them

            for (int i = 0; i < elevators.size(); i++) {

                if (elevators.get(i).canHandle(p)) {

                    elevators.get(i).push(p);

                    placed = true;

                    break;

                }

            }

            // If no available elevator could handle the person, open a new one

            if (!placed) {

                elevators.add(new ElevatorBin(maxLoad, maxArea, p));

            }

        }

    }

    // Best Fit Algorithm execution

    public void bestFit(ElevatorQueue people) {

        // Add the first elevator into elevators

        elevators.add(new ElevatorBin(maxLoad, maxArea));

        for (Person p : people) {

            // If the person is too big for the elevator, increment cannotFit and continue

            if (p.getArea() > maxArea || p.getWeight() > maxLoad) {

                cannotFit++;

                continue;

            }

            // The available elevators that can accept the person

            ElevatorArrayList<Integer> available = new ElevatorArrayList<>();

            // Search for all available elevators

            for (int i = 0; i < elevators.size(); i++) {

                if (elevators.get(i).canHandle(p)) {

                    available.add(i);

                }

            }

            // Open a new elevator to accept the person if no available elevator can

            if (available.isEmpty()) {

                elevators.add(new ElevatorBin(maxLoad, maxArea, p));

            }

            // Else put the person into the elevator with the least space remaining

            else {

                // initial index of available elevators

                int min = 0;

                // Search for index of available elevators with the least space remaining

                for (int i = 0; i < available.size(); i++) {

                    if (elevators.get(available.get(i)).getCurrentLoad() > elevators.get(available.get(min))

                            .getCurrentLoad()) {

                        min = i;

                    }

                }

                elevators.get(available.get(min)).push(p);

            }

        }

    }

    // Next Fit Algorithm execution

    public void nextFit(ElevatorQueue people) {

        // Add the first elevator into elevators

        elevators.add(new ElevatorBin(maxLoad, maxArea));

        // Index of the currently used elevator

        int current = 0;

        for (Person p : people) {

            // If the person is too big for the elevator, increment cannotFit and continue

            if (p.getArea() > maxArea || p.getWeight() > maxLoad) {

                cannotFit++;

                continue;

            }

            // If the elevator can accept the person, add the person

            if (elevators.get(current).canHandle(p)) {

                elevators.get(current).push(p);

            }

            // Else open a new elevator to accept the person

            else {

                current++;

                elevators.add(new ElevatorBin(maxLoad, maxArea, p));

            }

        }

    }

    // Worst Fit Algorithm execution

    public void worstFit(ElevatorQueue people) {

        // Add the first elevator into elevators

        elevators.add(new ElevatorBin(maxLoad, maxArea));

        for (Person p : people) {

            // If the person is too big for the elevator, increment cannotFit and continue

            if (p.getArea() > maxArea || p.getWeight() > maxLoad) {

                cannotFit++;

                continue;

            }

            // The available elevators that can accept the person

            ElevatorArrayList<Integer> available = new ElevatorArrayList<>();

            // Search for all available elevators

            for (int i = 0; i < elevators.size(); i++) {

                if (elevators.get(i).canHandle(p)) {

                    available.add(i);

                }

            }

            // Open a new elevator to accept the person if no available elevator can

            if (available.isEmpty()) {

                elevators.add(new ElevatorBin(maxLoad, maxArea, p));

            }

            // Else put the person into the elevator with the most space remaining

            else {

                // initial index of available elevators

                int max = 0;

                // Search for index of available elevators with the most space remaining

                for (int i = 0; i < available.size(); i++) {

                    if (elevators.get(available.get(i)).getCurrentLoad() < elevators.get(available.get(max))

                            .getCurrentLoad()) {

                        max = i;

                    }

                }

                elevators.get(available.get(max)).push(p);

            }

        }

    }

}

## 7.9 AlgorithmTest.java (Main Program)

import java.util.Scanner;

public class AlgorithmTest {

    public static void main(String[] args) {

        long startTime = 0;

        Scanner input = new Scanner(System.in);

        boolean running = true;

        // Create a queue of people from the file

        ElevatorQueue people = new ElevatorQueue();

        // Load the unique people from People.txt

        people.load("People.txt");

        System.out.println("There is " + people.getSize() + " unique records in the queue of people.\n");

        // Loop until the user decides to exit

        while (running) {

            // Get the specification of the elevator from user

            System.out.println("Please enter the specifications of the elevator");

            System.out.print("Max Elevator Load (kg): ");

            // Input for elevator capacity

            double elevatorCapacity = 0;

            while (true) {

                if (input.hasNextDouble()) {

                    elevatorCapacity = input.nextDouble();

                    if (elevatorCapacity > 0) {

                        break;

                    } else {

                        System.out.print("Please enter a positive number: ");

                    }

                } else {

                    System.out.print("Please enter a valid number: ");

                    input.next();

                }

            }

            // Input for elevator area

            System.out.print("Max Elevator Area (m^2): ");

            double elevatorArea = 0;

            while (true) {

                if (input.hasNextDouble()) {

                    elevatorArea = input.nextDouble();

                    if (elevatorArea > 0) {

                        System.out.println("");

                        break;

                    } else {

                        System.out.print("Please enter a positive number: ");

                    }

                } else {

                    System.out.print("Please enter a valid number: ");

                    input.next();

                }

            }

            // Get the algorithm to be tested from user

            System.out.println("Please select the algorithm to be tested");

            System.out.println(

                    "1. First Fit\n2. Best Fit\n3. Next Fit\n4. Worst Fit\n5. All Algorithms");

            System.out.print("Algorithm (1-5): ");

            int algoChoice = 0;

            while (true) {

                if (input.hasNextInt()) {

                    algoChoice = input.nextInt();

                    if (algoChoice >= 1 && algoChoice <= 5) {

                        System.out.println("");

                        break;

                    } else {

                        System.out.print("Please enter a number between 1 and 5: ");

                    }

                } else {

                    System.out.print("Please enter a valid number: ");

                    input.next();

                }

            }

            // Executing the selected algorithm

            String[] algoSelected = null;

            switch (algoChoice) {

                case 1:

                    algoSelected = new String[] { "First Fit" };

                    break;

                case 2:

                    algoSelected = new String[] { "Best Fit" };

                    break;

                case 3:

                    algoSelected = new String[] { "Next Fit" };

                    break;

                case 4:

                    algoSelected = new String[] { "Worst Fit" };

                    break;

                case 5:

                    algoSelected = new String[] { "First Fit", "Best Fit", "Next Fit", "Worst Fit" };

                    break;

            }

            Algorithms singleAlgo = null;

            for (String algo : algoSelected) {

                startTime = System.currentTimeMillis();

                Algorithms algorithm = new Algorithms(algo, elevatorCapacity, elevatorArea, people);

                algorithm.getResult();

                printRuntime(startTime);

                singleAlgo = algorithm;

            }

            // Ask if the user wants to see the details of each elevator

            if (algoChoice != 5) {

                System.out.println("Would you like to see the details of each elevator bin?");

                System.out.println("Note: Not recommended if large number of elevators is used");

                System.out.print("Show bins (y/n): ");

                String choice = "";

                while (true) {

                    choice = input.next().toLowerCase();

                    if (choice.equals("y") || choice.equals("n")) {

                        System.out.println("");

                        break;

                    } else {

                        System.out.print("Please enter y or n: ");

                    }

                }

                if (choice.equals("y")) {

                    singleAlgo.getBins();

                }

            }

            // Ask if the user wants to test another algorithm

            System.out.println("Would you like to test another algorithm?");

            System.out.print("Continue (y/n): ");

            String continueChoice = "";

            while (true) {

                continueChoice = input.next().toLowerCase();

                if (continueChoice.equals("y") || continueChoice.equals("n")) {

                    System.out.println("");

                    break;

                } else {

                    System.out.print("Please enter y or n: ");

                }

            }

            if (continueChoice.equals("n")) {

                running = false;

            } else {

                System.out.print("\033[H\033[2J");

                System.out.flush();

            }

        }

        // Exit the program

        System.out.println("Thank you for using the Elevator Packing Algorithm Tester!");

        input.close();

    }

    public static void printRuntime(long startTime) {

        System.out.println("Time taken (ms): " + (System.currentTimeMillis() - startTime) + "\n");

    }

}

## 7.10 PersonGenerator (Bonus Program)

import java.io.FileWriter;

import java.io.IOException;

import java.text.DecimalFormat;

import java.util.ArrayList;

import java.util.Random;

// IMPORTANT: this is only used to generate the data for the people.txt file and is not part of the main program

public class PersonGenerator {

    public static void main(String[] args) {

        Random rand = new Random(727);

        double area = 0;

        double weight = 0;

        ArrayList<Person> people = new ArrayList<>();

        DecimalFormat numberFormat = new DecimalFormat("#0.00");

        try (FileWriter writer = new FileWriter("People.txt")) {

            for (int i = 0; i < 100000; i++) {

                area = rand.nextDouble(0.11) + 0.15;

                weight = rand.nextDouble(101) + 30;

                if (people.contains(new Person(area, weight))) {

                    i--;

                    continue;

                }

                people.add(new Person(area, weight));

                writer.write("\n" + numberFormat.format(area) + " " + numberFormat.format(weight));

            }

            System.out.println("100000 records of unique person has been created successfully");

            writer.close();

        } catch (IOException e) {

            System.out.println("Failed to generate people: " + e.getMessage());

        }

    }

}