Course Project

CMPS323 - Design and Analysis of Algorithms

Dr. Mohammad Saleh Mustafa Saleh

Eng. Alaa Sayed Ahmed Hussein

Fahrel Azki Hidayat – 202206836

Marcus Wein Monteiro – 202206892

# Proposed Solution

In this section, we will outline the solution description and implementation details of our chosen approach for optimizing elevator operations. We have employed a greedy algorithm to tackle the problem, which is detailed comprehensively in terms of its design and functionality. Additionally, we will discuss how this greedy approach effectively addresses the constraints and objectives specified in the problem statement.

**Approach**

Greedy Approach was utilized for a simpler implementation and to have better computational speed without losing significant quality to the solutions solved for moderate-sized and less complex situations of optimizing elevator operations. This approach involves selecting the current most optimal floor to stop at, only considering the passengers of requested floors in which their cost of walking to a previously calculated best floor was not calculated. Meanwhile, also minimizing the number of stops and floors to ascend.

**Algorithm Detail**

* First, computes the walking cost of passengers to their respective requested floors in the set S (A set of requested floors where their walking cost has not been computed from a previously calculated optimal floor. Initially, this set includes all the requested floors) from the currently checked floor in the set of unvisited floors known as U ranging from the lowest requested floor to the highest requested floor. The walking cost is the product of the number of people requested for floor *f* and the distance between the unvisited floor *u* and floor *f*. While computing all the floors walking cost in the set of unvisited floors U, it keeps track of the floor that has the minimum accumulating walking cost to the requested floors. After computing all the floors in U, one floor in U will be stored as the locally optimal floor *o* as one of the stops.
* **SPECIAL CASE:** Although, for large problem sizes with complex scenarios in which there exists outliers in the higher floors. The algorithm will adjust the walking cost by doubling the cost from a floor *u* in set U to the high-level requested floor *h* (outlier). This results in considering the higher floor as one of the optimal floor *o* to stop at.
* Next, after finding the optimal floor *o* to stop. The floor *o* will be the stop for one of the requested floors in the Set S, the floor chosen will be the closest requested floor *f* to the calculated floor *o*. Thus, the locally optimal stop for floor *f* will be floor *o*. Afterwards, the walking cost from floor *f* to floor *o* is computed and added to a variable that accumulates the cost (totalCost). Then, it removes the floor *f* from the set S and floor *o* is marked as visited and removed from the set U so they are not considered during the next calculation of finding the locally optimal floor. Finally, add the floor *o* to set B which consists of optimal floors that has been calculated previously (Initially, the set B is empty) and set B will be the solution to the best floors to stop at with k stops.
* At the end of the above 2 steps, the value k (stops) will be decremented by 1 and the steps are done repeatedly until the number of calculated locally optimal stops reaches the number of allowed stops or the set S becomes empty.
* If the value of k reaches 0 meaning it has used the maximum number of stops, the solution set B which consists of the best floors to stop at is finalized. Although the set S may not be empty implying that the remaining requested floors in set S walking costs have not been calculated. Thus, computing the walking cost of each remaining requested floors *r* in set S is done by finding the closest stopped floor *c* at set B to the floor *r* then calculate the walking cost between floor *c* and floor *r*.

**Constraint & Objective Handling**

* **Minimizing the Total Walking Cost** is acquired by choosing the stops *o*’s in the set of unvisited floors which is set U that has the minimum accumulating walking distance to reach the requested floors in set S. Then choosing a requested floor *f* in set S where its closest to the locally optimal stop *o* in set U. Thus, ensuring the distance is reduced for the riders to reach their requested floor *f*.
* **The Tie-Breaking rule** is evaluated when two or more unvisited floors in set U have an equal accumulating walking distance to reach the requested floors in set S. In this case, the algorithm will prefer taking the lower floor as the locally optimal stop *o*. This is secured by computing the unvisited floors in set U in ascending order and a new minimum accumulating walking distance only occurs when the cost of a currently assessed unvisited floor in set U is **less** than the cost of the optimal stop so far.
* The algorithm adheres to theconstraint of the **Maximum stops granted**. This guarantees that the number of appointed stops will not surpass the allowed maximum limit. Thus, the stop selected for the remaining requested floors *f*’sin set S which have not been determined will be the closest floor *c*’sin set B. Therefore, it optimizes the walking cost for the riders to arrive at their respective requested floors *f*.

# Solution

In this section, we detail the implementation of our proposed solution for optimizing elevator operations. We will delve into the specific data structures and algorithms employed in our greedy algorithm approach. Moreover, the discussion will cover any external tools and libraries utilized. A complete copy of the source code will also be provided, offering a thorough examination of how each component of the solution integrates with each other to achieve the desired greedy algorithm.

**Class Implementation with its attributes and methods**

The **ElevatorOptimization** Class is the only class developed to construct the greedy algorithm for the elevator optimization problem.

**ElevatorOptimizations Attributes/Instance Variables:**

* **int** *numOfStops* – The upper limit for the number of stops permitted.
* **int[]** *floorArray –* The array stores a list of requested floors.
* **int[]** *peoplePerFloorArray* – The array stores the weight (number of people) of the corresponding index of the floors in the **int[]** *floorArray*. (An implementation of Set S)
* **ArrayList<Integer>** *stoppedFloorArray* – The ArrayList stores a list of locally optimal stopped floors. (An implementation of Set B)
* **int[]** *unvisitedFloorArray* – The array length is the difference between highest and lowest destination floors + 1 and stores 0 in the index of unvisited floors, otherwise stores 1 for visited floors. (An implementation of Set U)

The array doesn’t display the number of the floor but the index does. To determine what index represents which floor we can use the formula:

**Visualizing Array Implementation with** *floorArray[0] = 5***:**

**Index:** 0 1 2

|  |  |  |
| --- | --- | --- |
| 0 | 0 | 1 |

**Using the formula:**

* At Index 0: 5 + 0 = 5th floor.
* At Index 1: 5 + 1 = 6th floor.
* At Index 2: 5 + 2 = 7th floor.

The **ElevatorOptimization** methods are classified into mainly three categories: the main greedy algorithm method, methods that supports the greedy algorithm functionand other methods that collects user input and performs validation checks. Also, the main method which executes the program.

**ElevatorOptimization Input/Data gathering, validation and display methods:**

* **void** *getData()* – This method utilizes a scanner object to prompt the user for inputs regarding the quantity of floors requested, the specific floor numbers and the number of passengers for each floor. It also includes validation checks to ensure that the input data for the three mentioned parameters is appropriate otherwise the user will be prompted to enter the data again until it meets the required criteria.
* **boolean** *isFloorRequested(***int[]***floors,***int** *floor,* **int** *currentIndex) –* It is a supportive method which acts as one of the validation check in *getData()* and returns false if the floor number is new otherwise it returns true. Its purpose is to prevent a previously provided floor number from being accepted if the same floor number is entered again.
* **void** *printList*(**int[]** *array)* – Another supportive method which takes the array **int[]** *floorArray* and **int[]** *peoplePerFloorArray* and prints them in a manner that makes it easy for the user to visualize the requested floors along with its weight (number of people).

**ElevatorOptimization Main Greedy Algorithm method:**

Step-by-step procedure of the **void** *optimizeGreedy()* leading to obtaining the optimal floors to stop at:

1. Before calling *optimizeGreedy()* the instance variables *numOfStops,* **int[]** *floorArray,* **int[]** *peoplePerFloorArray* has been initialized and filled with the data received from user input and the arrays are sorted in ascending order using *sortArrays()* which is vital for this algorithm to function and meet the requirements of the tie-breaking rule of the proposed solution. It is a supportive method of the greedy algorithm mentioned below this section. Also, **ArrayList<Integer>** *stoppedFloorArray* is initialized but empty.
2. The instance variable **int[]** *unvisitedFloorArray* can be created now with the size of . Highest requested floor and lowest requested floor is obtained by [floorArray[floorArray.length - 1] and floorArray[0], respectively. In addition, a local variable totalCost is initialized to 0 which will consist of the sum of the walking cost after the end of the algorithm.

**Inside the while loop that runs until** *numOfStops* **= 0 (Outer while-loop):**

1. Intializes local variables currentMinCost and currentMinFloor with Highest possible integer value in Java and floorArray[0] which is the lowest requested floor, respectively. They are used to track the unvisited floors that have the minimum total walking cost to all destination floors in which their cost has not been computed using a previously locally optimal floor to stop at.

**Inside the for loop that runs until** **all unvisited floors walking cost is calculated (First Inner for-loop):**

1. First, an **IF** function checks the current index in the first inner loop meaning the current floor being assessed in the array *unvisitedFloorArray*. If *unvisitedFloorArray*[index/floor] = 1, floor is visited and in *stoppedFloorArray* meaning it is already a solution in the set of optimal stopped floors so skip the calculation of that floor. Otherwise, if *unvisitedFloorArray*[index/floor] = 0, it does not go inside the **IF** function.
2. Initializes the currentCost = 0. This tracks the walking cost of one of the currently evaluated unvisited floors.

**Inside the for loop that runs until the walking cost to reach all requested floor from the currently checked unvisited floor is computed (Second Inner for-loop)**

1. Initially, an **IF** function checks if the cost to reach the requested floor has been calculated using a previously calculated locally optimal stop floor. If *peoplePerFloorArray*[j] = 0, then the minimized cost to reach that requested floor has been calculated so skip the calculation for that floor. Otherwise if *peoplePerFloorArray*[j] != 0, it does not go inside the **IF** function.
2. Then, difference in levels between the currently checked requested floor and unvisited floor is calculated (A) using absolute value function to avoid negative values. Next, the cost of walking to the requested floor is calculated by the formula:
3. Before accumulating the currentCost to totalCost, the supportive method **int** *adjustCostForSparseHighRanges*(**int** *floor*, **int** *distance*) is called to modify the walking cost between the currently checked unvisited floor and requested floor in order to address the possibility of stopping at higher floors (outliers) without disregarding them. Then accumulates the cost to currentCost.
4. Continues executing steps 6 and 7, 8 until all the cost to reach the requested floors from the unvisited floors has been computed.

**End of Second Inner for-loop**

1. An **IF** functions check if the walking cost of the currently examined unvisited floor to reach all the requested floors with value of *peoplePerFloorArray*[j] != 0 is **less** than the currentMinCost meaning its less than the optimal unvisited floor so far, then the currently unvisited floor cost and floor number becomes the new currentMinCost and currentMinFloor, respectively (**Selection Procedure and Feasibility Check**)

**End of First Inner for-loop**

1. An **int[]** *closestFloorAndCost* is initialized to the return value of a supportive method *findClosestFloor(***int** *currentMinFloor)*. *closestFloorAndCost[0]* consists of the closest requested floor tocurrentMinFloor and *closestFloorAndCost[1]* contains the cost from the currentMinFloor to the closest requested floor and accumulates it in totalCost.
2. The supportive method *getIndex(***int** *floor,* **int[]** *arr)* returns the index of the closest requested floor to currentMinFloor. Then, *peoplePerFloorArray*[index] will now be set to 0 indicating that the walking cost to reach requested floor of that index in *peoplePerFloorArray* has been calculated using currentMinFloor which the current locally optimal floor to stop at. (In other words, the removal of the requested floor from the set S)
3. Then the index of currentMinFloor in the *unvisitedFloorArray* is calculated by . Afterwards, *unvisitedFloorArray[index]* is set to 1 implying that the unvisited floor which is in the variable currentMinFloor has been visited. In addition, the currentMinFloor meaning the current locally optimal floor to stop at is appended to *stoppedFloorArray*. (In other words, the floor in the variable currentMinFloor is removed from the set of unvisited floors U and added into set B which contains the solution of floors to stop at).
4. After steps 3 to 13, the instance variable *numOfStops* is decremented by 1 and the steps 3 to 13 is repeatedly executed until *numOfStops* = 0. (**Solution Check**)

**End of Outer while-loop**

1. Sorts the *stoppedFloorArray* Array List using the sort method form Class Collections arranging it in ascending order.
2. The supportive method *remainingCost()* returns the value of the total cost to reach the requested floors that has not been calculated in the loops because *numOfStops* reaches 0 before it can be computed. This value is accumulated to the variable totalCost.
3. Prints the result of all the floors in *stoppedFloorArray* which are calculated to be the optimal floors to stop at as well as the total cost to reach the requested floors from the floors in *stoppedFloorArray*.

**ElevatorOptimization Greedy Algorithm supportive methods:**

* **int[]** *findClosestFloor***(int** *currentMinFloor***)** – This supportive method searches for the closest requested floor in the *floorArray* to the currentMinFloor but ignores the consideration of requested floors having *peoplePerFloorArray*[floor] = 0 since their walking cost has already been computed with a previously local optimal stop floor. It returns the floor number closest to the currentMinFloor and the walking cost from it to the currentMinFloor.
* **int** *remainingCost()* – This supportive method returns the total cost of all the remaining requested floors in *floorArray* where the *peoplePerFloorArray*[floor] != 0 meaning that the walking cost to that requested floor has not been calculated with a locally optimal floor to stop at.
* **int** *getIndex*(**int** *value*, **int[]** *arr*) – The supportive method in this greedy algorithm takes the *floorArray* and the requested floor value in order to return its index.
* **void** *sortArrays()* and  *merge()* – Both supportive methods are used to sort the *floorArray* and *peoplePerFloorArray* in ascending order while maintaining so that each corresponding values in *floorArray* and *peoplePerFloorArray* represents that the requested floor has n people. Crucial to adhere to the tie-breaking rule of this algorithm.
* **int** *adjustCostForSparseHighRanges*(**int** *floor*, **int** *distance*) – This supportive method takes the parameter **int** floor which is a requested floor and **int** distance which is the normal walking cost to reach the requested floor before modification. It ensures to not rule out the higher floors as a potential floor to stop at. The method determines whether a requested floor is at high-level by examining if it’s greater than the defined threshold below:

Value A specifies the amount of floors that are classified as high-level floors. The Threshold is the difference between the highest requested floor and A forming a specific range of floors that are recognized as a high-level floor.

If, this requested floor is greater than the threshold meaning it’s a high-level floor. The walking cost (**int** distance) between the currently assessed unvisited floor and requested floor will be doubled. Otherwise, the walking cost remains as it is. Finally, returns the new walking cost if modified.

**Data Structures, and libraries employed during implementation**

Certain data structures and libraries were utilized in order to facilitate and construct the proper greedy algorithm.

**Data Structures utilized:**

* **ArrayList** was used to create *stoppedFloorArray* in order to take advantage of the class’s method *add(*Object o*)* for inserting the locally optimal floor to stop at each step efficiently.
* **Array** was applied frequently to store data with fixed sizes such as *floorArray*, *peoplePerFloorArray* and *unvisitedFloorArray*. Since, we don’t perform any insertion or deletion but only modification in the values of this array. We can exploit the indexing operation in arrays to change their values effectively.

**Libraries Imported and others:**

* **ArrayList** is imported for initializing **ArrayList<Integer>** *stoppedFloorArray*.
* **Collections** library is utilized to sort **ArrayList<Integer>** *stoppedFloorArray* in ascending order.
* **Scanner** package is used to collect data from user input.
* **Wrapper** class is used to initialize the ArrayList with type **Integer**.

**Java Source Code**

**package** project;

**import** java.util.Scanner;

**import** java.util.ArrayList;

**import** java.util.Collections;

**public** **class** ElevatorOptimization {

**public** **static** **int** *numOfStops*;

**public** **static** **int**[] *floorArray*;

**public** **static** **int**[] *peoplePerFloorArray*;

**public** **static** ArrayList<Integer> *stoppedFloorArray* = **new** ArrayList<>();

**public** **static** **int**[] *unvisitedFloorArray*;

**public** **static** **void** main(String[] args) {

ElevatorOptimization elevator = **new** ElevatorOptimization();

elevator.getData();

System.***out***.println();

System.***out***.print("Floors requested: ");

*printList*(*floorArray*);

System.***out***.print("Number of People Each Floor: ");

*printList*(*peoplePerFloorArray*);

*optimizeGreedy*();

}

// Prints the floors it stops in along with the total cost of walking.

**private** **static** **void** optimizeGreedy() {

**if** (*stoppedFloorArray* == **null**) {

*stoppedFloorArray* = **new** ArrayList<>(); // Safety check

}

// Shows 0 if it has not stopped at that floor otherwise Shows 1. Shows floor between the lowest and highest floor requested.

*unvisitedFloorArray* = **new** **int**[(*floorArray*[*floorArray*.length - 1] - *floorArray*[0]) + 1];

// Initializing Total cost of walking up or down the stairs.

**int** totalCost = 0;

// Loop is Finding all the best floor to stop at.

**while** (*numOfStops* != 0) {

// Initializing The floor with the minimum cost of walking.

**int** currentMinCost = Integer.***MAX\_VALUE***;

**int** currentMinFloor = *floorArray*[0];

// The Loop Finds the current best floor to stop at. (Takes the locally optimal floor).

**for** (**int** i = 0; i < *unvisitedFloorArray*.length; i++) {

// Skips calculating that floor if it is already visited/stopped.

**if** (*unvisitedFloorArray*[i] == 1)

**continue**;

**int** currentCost = 0;

**for** (**int** j = 0; j < *floorArray*.length; j++) {

// Skips calculating the floor in which its cost has been calculated in Line 80.

**if** (*peoplePerFloorArray*[j] == 0) {

**continue**;

}

// Difference in levels between requested floor and currently checked floor.

**int** differenceBetweenFloor = Math.*abs*(*floorArray*[0] + i - *floorArray*[j]);

**int** costOfWalkingToFloorI = differenceBetweenFloor \* *peoplePerFloorArray*[j];

costOfWalkingToFloorI += *adjustCostForSparseHighRanges*(*floorArray*[j], differenceBetweenFloor);

currentCost += costOfWalkingToFloorI;

}

// if the currently checked floor has the minimum total cost of all the

// requested Floors in which its cost has not been calculated in Line 80 it

// becomes the current best floor for now.

**if** (currentCost < currentMinCost) {

currentMinCost = currentCost;

currentMinFloor = *floorArray*[0] + i;

}

}

// Finds the closest requested Floor to the current best stopped floor

// (currentMinFloor) and calculates its

// walking cost to the current best stopped floor (currentMinFloor) and adds to

// the totalCost

**int**[] closestFloorAndCost = *findClosestFloor*(currentMinFloor);

**int** index = *getIndex*(closestFloorAndCost[0], *floorArray*);

totalCost += closestFloorAndCost[1];

// Sets the closest Floor weight (no. of people) to 0 to show that its cost has been calculated.

*peoplePerFloorArray*[index] = 0;

// Finds position of the best stopped floor (currentMinFloor) in the

// unvistedArrayFloor and convert its

// index to 1 showing that it is visited/stopped.

**int** position = currentMinFloor - *floorArray*[0];

*unvisitedFloorArray*[position] = 1;

// Adds the current best floor to stop at.

*stoppedFloorArray*.add(currentMinFloor);

// Decrements number of stops since one stopped Floor has been calculated.

*numOfStops*--;

}

// Sorts the stopped Floors

Collections.*sort*(*stoppedFloorArray*);

totalCost += *remainingCost*();

// Prints result.

System.***out***.println("Stopped Floors: " + *stoppedFloorArray*);

System.***out***.println("Total Cost: " + totalCost);

}

// Adjusting cost for high-range floors, considering them more strategically if they are distant

**private** **static** **int** adjustCostForSparseHighRanges(**int** floor, **int** distance) {

// Threshold for considering a floor "high-range"

**int** threshold = (*floorArray*[*floorArray*.length - 1] - *floorArray*[0]) / 4;

**if** (floor > *floorArray*[*floorArray*.length - 1] - threshold) {

**return** distance \* 2; // Increasing the cost penalty for floors beyond the threshold

}

**return** 0;

}

**private** **static** **int**[] findClosestFloor(**int** currentMinFloor) {

**int** closestStopDistance = Integer.***MAX\_VALUE***;

**int** closestFloor = -1;

**for** (**int** i = 0; i < *floorArray*.length; i++) {

**if** (*peoplePerFloorArray*[i] == 0)

**continue**;

**int** distance = Math.*abs*(currentMinFloor - *floorArray*[i]);

**if** (distance < closestStopDistance) {

closestStopDistance = distance;

closestFloor = *floorArray*[i];

}

}

**return** **new** **int**[] { closestFloor, closestStopDistance };

}

**private** **static** **int** getIndex(**int** value, **int**[] arr) {

**for** (**int** i = 0; i < arr.length; i++) {

**if** (arr[i] == value) {

**return** i;

}

}

**return** -1;

}

**private** **static** **int** remainingCost() {

**int** totalCost = 0;

**for** (**int** i = 0; i < *floorArray*.length; i++) {

**if** (*peoplePerFloorArray*[i] == 0) {

**continue**;

}

**int** floor = *floorArray*[i];

**int** closestStopDistance = Integer.***MAX\_VALUE***;

**for** (**int** stop : *stoppedFloorArray*) {

**int** distance = Math.*abs*(stop - floor);

**if** (distance < closestStopDistance) {

closestStopDistance = distance;

}

}

**int** floorCost = closestStopDistance \* *peoplePerFloorArray*[i];

totalCost += floorCost;

}

**return** totalCost;

}

// Asks user input for requested floors and the number of people that requested

// for it.

**private** **void** getData() {

Scanner scanner = **new** Scanner(System.***in***);

// Prompt and validate the number of floors requested

System.***out***.println();

System.***out***.print("Enter the number of floors requested: ");

**int** numOfFloors = scanner.nextInt();

**while** (numOfFloors <= 0) {

System.***out***.println("The number of floors must be greater than zero. Please enter a valid number: ");

numOfFloors = scanner.nextInt();

}

System.***out***.println();

*floorArray* = **new** **int**[numOfFloors];

*peoplePerFloorArray* = **new** **int**[numOfFloors];

**for** (**int** i = 0; i < numOfFloors; i++) {

System.***out***.print("Enter the requested floor (Request No.: " + (i + 1) + "): ");

**int** floorRequest = scanner.nextInt();

// Validate the floor request is not zero

**while** (floorRequest <= 0) {

System.***out***.println();

System.***out***.print("Floor number must be greater than zero. Please enter a valid floor: ");

floorRequest = scanner.nextInt();

}

// Check if the floor has already been requested by searching the existing

// entries in the array

**while** (isFloorRequested(*floorArray*, floorRequest, i)) {

System.***out***.println();

System.***out***.print("This floor has already been requested. Please enter a different floor: ");

floorRequest = scanner.nextInt();

}

*floorArray*[i] = floorRequest;

System.***out***.println();

System.***out***.print("Enter the number of people requested for floor " + *floorArray*[i] + ": ");

*peoplePerFloorArray*[i] = scanner.nextInt();

// Validate the number of people is not zero

**while** (*peoplePerFloorArray*[i] <= 0) {

System.***out***.println();

System.***out***.println("The number of people must be greater than zero. Please enter a valid number: ");

*peoplePerFloorArray*[i] = scanner.nextInt();

}

System.***out***.println();

}

*sortArrays*(numOfFloors);

System.***out***.print("Enter the number of stops: ");

*numOfStops* = scanner.nextInt();

System.***out***.println();

// Ensure that the number of stops is greater than zero and does not exceed the

// number of unique floors requested

**while** (*numOfStops* <= 0 || *numOfStops* > numOfFloors) {

**if** (*numOfStops* <= 0) {

System.***out***.println("The number of stops must be greater than zero. Please enter a valid number:");

} **else** {

System.***out***.println(

"The number of stops cannot exceed the number of floors requested. Please enter a valid number of stops:");

}

*numOfStops* = scanner.nextInt();

}

scanner.close();

}

// Check if a floor has already been requested

**private** **boolean** isFloorRequested(**int**[] floors, **int** floor, **int** currentIndex) {

**for** (**int** j = 0; j < currentIndex; j++) {

**if** (floors[j] == floor) {

**return** **true**;

}

}

**return** **false**;

}

// Organizes the two arrays so that each corresponding value means the Requested

// Floor has n people.

**public** **static** **void** sortArrays(**int** size) {

**if** (size < 2) {

**return**; // No need to sort

}

**int** mid = size / 2;

**int**[] leftFloors = **new** **int**[mid];

**int**[] rightFloors = **new** **int**[size - mid];

**int**[] leftPeople = **new** **int**[mid];

**int**[] rightPeople = **new** **int**[size - mid];

// Dividing the arrays into two halves

**for** (**int** i = 0; i < mid; i++) {

leftFloors[i] = *floorArray*[i];

leftPeople[i] = *peoplePerFloorArray*[i];

}

**for** (**int** i = mid; i < size; i++) {

rightFloors[i - mid] = *floorArray*[i];

rightPeople[i - mid] = *peoplePerFloorArray*[i];

}

*sortArrays*(leftFloors, leftPeople, mid);

*sortArrays*(rightFloors, rightPeople, size - mid);

// Merging the sorted halves

*merge*(*floorArray*, *peoplePerFloorArray*, leftFloors, rightFloors, leftPeople, rightPeople, mid, size - mid);

}

// Recursive sort function

**private** **static** **void** sortArrays(**int**[] floors, **int**[] people, **int** size) {

**if** (size < 2) {

**return**;

}

**int** mid = size / 2;

**int**[] leftFloors = **new** **int**[mid];

**int**[] rightFloors = **new** **int**[size - mid];

**int**[] leftPeople = **new** **int**[mid];

**int**[] rightPeople = **new** **int**[size - mid];

**for** (**int** i = 0; i < mid; i++) {

leftFloors[i] = floors[i];

leftPeople[i] = people[i];

}

**for** (**int** i = mid; i < size; i++) {

rightFloors[i - mid] = floors[i];

rightPeople[i - mid] = people[i];

}

*sortArrays*(leftFloors, leftPeople, mid);

*sortArrays*(rightFloors, rightPeople, size - mid);

*merge*(floors, people, leftFloors, rightFloors, leftPeople, rightPeople, mid, size - mid);

}

// Merging the sorted arrays back together

**private** **static** **void** merge(**int**[] floors, **int**[] people, **int**[] leftFloors, **int**[] rightFloors, **int**[] leftPeople,

**int**[] rightPeople, **int** left, **int** right) {

**int** i = 0, j = 0, k = 0;

**while** (i < left && j < right) {

**if** (leftFloors[i] <= rightFloors[j]) {

floors[k] = leftFloors[i];

people[k] = leftPeople[i];

i++;

} **else** {

floors[k] = rightFloors[j];

people[k] = rightPeople[j];

j++;

}

k++;

}

**while** (i < left) {

floors[k] = leftFloors[i];

people[k] = leftPeople[i];

i++;

k++;

}

**while** (j < right) {

floors[k] = rightFloors[j];

people[k] = rightPeople[j];

j++;

k++;

}

}

**private** **static** **void** printList(**int**[] array) {

System.***out***.print("[");

**for** (**int** i = 0; i < array.length; i++) {

System.***out***.print(array[i]);

**if** (i < array.length - 1) {

System.***out***.print(", ");

}

}

System.***out***.println("]");

}

}

# Experimental Results and Screenshots

In this section, we will detail the methods employed to gather and analyze both quantitative and qualitative data regarding the algorithm's performance. Here, we will showcase various trial scenarios, discussing each quantitatively and qualitatively to provide a comprehensive understanding of the results. This analysis will provide interpretations to assess the practical implications of the optimization under different conditions, reflecting on how effectively our proposed algorithm meets the designated performance criteria.

**Experimental Setup**

The experimental setup of the elevator optimization code involves simulating the elevator request scenarios to evaluate the effectiveness and efficiency of the algorithm. The primary function facilitating this setup is the getData() method, which collects user inputs necessary for the experiment. Below is a detailed description of how this function contributes to setting up the experiment.

**Functionality of getData method**

* **Collect Number of Floors Requested**
  + The function prompts the user to enter the total number of floor requests and validates if the entered value is greater than zero.
* **Request Specific Floors and People Count**
  + For each floor request, the function asks for two specific pieces of information:
    - The floor number, ensuring that each entered floor is a positive integer.
    - The number of people who have requested that specific floor, also ensuring this number is positive.
  + This step is crucial as it mimics practical scenarios where different floors may have varying numbers of people waiting, affecting the decision of the greedy algorithm.
* **Input for Number of Stops**
  + After gathering all floor requests, the function requests the number of stops the elevator is allowed to make. This number must not exceed the number of unique floors requested and must also be a positive integer.

**Implementing the sortArrays Method**

* In addition to the earlier description of the getData() function, the sortArrays method plays a crucial role in presenting the data for the experimental setup.
* It is designed to organize the input data collected via the getData() method. It sorts the arrays using **merge sort** that store the floor numbers (floorArray) and the corresponding number of people requesting each floor (peoplePerFloorArray) in ascending order based on the floor numbers.

**Usage of the remainingCost method**

* In the experimental setup, the remainingCost() function is essential for quantitatively assessing the effectiveness of the elevator optimization algorithm. Calculating the total walking cost is invaluable as it provides a direct metric to evaluate how well the implemented optimization strategy reduces passenger inconvenience.

**Experimental Results**

We present the experimental results from testing our algorithm across six different instances, evaluating its performance under various conditions. Each test case details setup specifics, such as floor requests, passenger distribution, and allowed stops, with quantitative and qualitative assessments comparing algorithm decisions to expected outcomes. This analysis illustrates the effectiveness and reliability of the greedy approach in unique scenarios.

Figure 1. Experimental Result of Trial 1

A computer screen shot of a number

Description automatically generated

**Quantitative Analysis of Trial 1**

* Floors Requested: [13, 14, 15]
* Number of People Each Floor: [1, 1, 1] where each floor has one person requesting a stop.
* Stopped Floors: [13, 14, 15] where the algorithm chose to stop at every requested floor.
* Number of Stops: 3, which matches the number of requested floors.
* Total Cost: 0, indicating that no passenger had to walk to another floor.

**Qualitative Analysis of Trial 1**

* The algorithm's greedy approach was ideally suited to the scenario, allowing a stop at each requested floor, and thus minimizing walking distances for all passengers, leading to an optimal performance.
* In this case, the alignment of the number of stops with the number of requests. This showcases the algorithm's effectiveness in providing optimal service under specific conditions where the number of floors requested is equal to the number of stopped floors, while the number of people are equal on each floor.

Figure 2. Experimental Result of Trial 2

A screenshot of a computer screen

Description automatically generated

**Quantitative Results of Trial 2**

* Floors Requested: [10, 15, 20, 30, 50]
* Number of People Each Floor: [3, 6, 1, 7, 3]
* Stopped Floors: [15, 30, 50] which are amid the requested range, minimizing the total distance passengers need to travel.
* Number of Stops: 3, which is less than the number of requested floors.
* Total Cost: 20

**Qualitative Analysis of Trial 2**

* The algorithm's choices reflect a tendency to **minimize the walking distance for the maximum number of people**. By choosing floors 15, 30, and 50, the algorithm effectively covered most passenger requests efficiently, particularly benefiting the larger groups on floors 15 and 30.
* The chosen stops suggest a prioritization of larger passenger groups, which could be seen as a fair compromise under the restriction of limited stops. Even if the total cost of 20 indicates that while the system is efficient in minimizing stops, it does not completely eliminate passenger inconvenience, particularly for those on unselected floors (10 and 20).

Figure 3. Experimental Result of Trial 3

A screenshot of a computer program

Description automatically generated

**Quantitative Results of Trial 3**

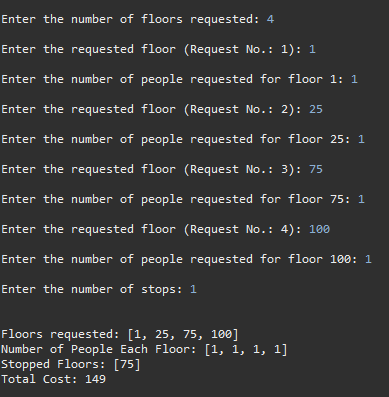
* Floors Requested: [34, 36]
* Number of People Each Floor: [4, 4]
* Stopped Floor: [34], which is the lower floor of the two requested floors.
* Number of Stops: 1, where we will account for the tie-breaker rule.
* Total Cost: 8

**Qualitative Analysis of Trial 3**

* The total cost of 8, although minimal, reflects the algorithm's success in accounting for the compromise made by allowing only one stop. This scenario tests the algorithm’s ability to decide in a tie-breaker situation.
* In a **tie-breaker situation**, the algorithm’s preference to stop at floor 34 instead of floor 36 suggests that it favors the lower floor. This preference aligns with management's strategy to reduce electricity consumption.

Figure 4. Experimental Result of Trial 4 (with unequal and equal floors)

A screenshot of a computer program

Description automatically generated 

**Quantitative Results of Trial 4 (unequal floors)**

* Floors Requested: [1, 25, 75, 100]
* Number of People Each Floor: [1, 1, 1, 3]
* Stopped Floors: [100], which has the highest demand of all the floors.
* Total Cost: 199

**Qualitative Analysis of Trial 4**

* The algorithm chose to stop only at floor 100, the highest requested floor, which had the most people waiting (3 people). However, it incurred a significant walking penalty for those who are on the lower floors. Given that the algorithm was constrained to make only one stop due to the limit set at one stop, it opted for the floor with the highest number of people.
* If in such scenarios where the number of passengers is equal across all floors, *if each floor had one passenger*, the algorithm would opt to stop at **floor 75, the upper median among the requested floors**, which is a strategic choice given the spread of the floors.

Figure 5. Experimental Result of Trial 5

A screenshot of a computer program

Description automatically generated

**Quantitative Results of Trial 5**

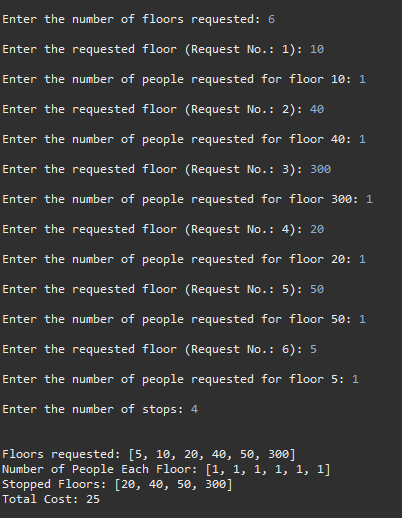
* Floors Requested: [3, 7, 10, 20, 25, 30, 200]
* Number of People Each Floor: [1, 1, 1, 1, 1, 1, 1]
* Stopped Floors: [10, 20, 25, 30, 200], which also includes the far outlier *floor 200.*
* Total Cost: 10

**Qualitative Analysis of Trial 5**

* This trial exhibits the importance of the **adjustCostForSparseHighRanges** function. In the trial results, stopping at floor 200 despite it being an extreme outlier demonstrates that the function effectively adjusts the perceived cost of not stopping at such a high range compared to the rest of the floors.
* The selection of floors 10, 20, 25, 30, and 200, with the omission of the nearest low floors (3 and 7), suggests a balancing act between minimizing total stops and reducing walking distances. This spread effectively covers the middle to high range, ensuring passengers on the omitted floors have access to nearby stops, albeit with some walking involved.

Figure 6. Experimental Result of Trial 6 (with unequal and equal floors)

A screenshot of a computer program

Description automatically generated 

**Quantitative Results of Trial 6 (unequal floors)**

* Floors Requested: [5, 10, 20, 40, 50, 300]
* Number of People Each Floor: [10, 3, 4, 2, 3, 1]
* Stopped Floors: [5, 10, 20, 50], which does not include the far outlier *floor 300.*
* Total Cost: 270

**Qualitative Analysis of Trial 6**

* The algorithm chose to stop at floors 5, 10, 20, and 50. This selection covers the floors with higher concentrations of passengers and omits the floor with the single passenger at the extreme outlier (floor 300). The stops cover the more densely populated lower floors.
* **Choosing not to stop at floor 300 can be seen as a limitation in terms of coverage** even if it makes sense from a demand perspective. Since the algorithm is designed to minimize total travel time and maximize service to the largest groups, stopping at a floor with only one passenger doesn't align with these goals, particularly under stop limitations.
* If in such scenarios where the number of passengers is equal across all floors, *if each floor had one passenger*, the algorithm would opt to stop at **floors [20, 40, 50, 300]**.Here, our algorithm also accounts for floor 300, since it ensures that the floor is not disproportionately disadvantaged by the algorithm's other choices.

# Discussion

**Time Complexity Analysis of the optimizeGreedy Method**

**Overall Time Complexity**

The overall time complexity of the optimizeGreedy method can be approximated as **O(k\*m\*n)** due to the nested loops involving the calculation of stopping costs, where:

* k is the number of stops,
* m is the range of floors, and
* n is the number of floor requests.

**Loops of Optimize Greedy**

* **Outer While-Loop**
  + while (numOfStops != 0)
* **First Inner For-Loop**
  + for (int i = 0; i < unvisitedFloorArray.length; i++)
* **Second Inner For-Loop**
  + for (int j = 0; j < floorArray.length; j++)

Figure 1. Critical Part of the optimizeGreedy Method

while(*numOfStops* != 0) { //loops up to k times

for (int i = 0; i < unvisitedFloorArray.length; i++) { // loops up to m times

if (unvisitedFloorArray[i] == 1) continue; // skip if already visited

int currentCost = 0;

for (int j = 0; j < *floorArray*.length; j++) { // loops n times

if (*peoplePerFloorArray*[j] == 0) {continue;}

// cost calculation here

int costOfWalkingToFloorI = differenceBetweenFloor \* *peoplePerFloorArray*[j];

costOfWalkingToFloorI += *adjustCostForSparseHighRanges*(*floorArray*[j],

differenceBetweenFloor);

currentCost += costOfWalkingToFloorI;

}

//calculations for calculating closest floor and indexing

}

}

**Implications of the Time Complexity of greedyOptimize Method**

In our algorithm, the complexity of O(k\*m\*n) might appear high, but it is essential to consider the problem's requirements and constraints. For many practical applications, especially where decisions aren't required in milliseconds, this complexity might be acceptable. This complexity can become quite large if any of these variables increases significantly, particularly in scenarios involving tall buildings with many floor requests and several stops allowed. However, in realistic elevator system applications, the number of floors (m) is generally finite and usually not exceedingly high. Moreover, the allowed stops per run (k) are fewer than the total floors, and the number of floor requests (n) per trip is limited. This reduces the practical workload for the algorithm, making a theoretically high complexity less of a concern in practical instances.

**Interpretation of Experimental Results**

The experimental results demonstrate the effectiveness of the greedy algorithm in optimizing elevator operations under various conditions:

* **Trial 1**: When the number of stops equaled the number of requested floors, the algorithm achieved zero total cost, indicating optimal alignment with the goal of minimizing walking distance.
* **Trial 2**: Facing unevenly distributed demands across many floors, the algorithm effectively chose stops to benefit floors with higher passenger densities, optimizing for general efficiency and convenience.
* **Trial 3**: With the tie-breaking rule in effect, the algorithm opted for the lower floor (34 instead of 36), demonstrating cost-efficiency by favoring easier-to-reach stops.
* **Trial 4**: Challenged by a wide range of floor requests and varying passenger numbers, the algorithm stopped only at the most populated floor (floor 100), revealing limitations in scenarios with restricted stops and diverse needs.
* **Trial 5**: In a densely populated setting, the algorithm chose stops across a broad spectrum of floors (10, 20, 25, 30, and 200), effectively minimizing walking for most but not for those on the lowest floors.
* **Trial 6**: Confronted with uneven passenger distribution, the algorithm skipped the least occupied floor (300) to focus on more populated stops (5, 10, 20, and 50), maximizing efficiency but showing limits in servicing less popular floors.

**Strengths of the Proposed Solution**

* The primary strength of the proposed solution lies in its **simplicity and computational efficiency**, which makes it suitable for practical applications in elevator systems.
* The greedy algorithm **adeptly accounts for the number of people and their demand on each floor**, prioritizing stops at floors with higher passenger concentrations. This strategy not only maximizes service efficiency but also adheres to the goal of minimizing overall walking costs.
* We have further refined the algorithm by incorporating the **adjustCostForSparseHighRanges()** method, specifically designed to reduce the likelihood of overlooking high floor outliers.
* Additionally, the algorithm's flexibility to adapt to various building sizes and passenger distributions without requiring significant modifications is a significant advantage.

**Limitations of the Proposed Solution**

* The greedy algorithm's **primary limitation is its nature of local optimization**. It makes decisions based on current, available information without forecasting future needs or consequences, which can prevent it from achieving globally optimal solutions.
* As demonstrated in Trial 4, the algorithm's focus on current conditions—**like servicing the floors with the highest numbers of passengers—can result in high total walking costs**. This occurs because it may neglect lower floors or upper floors with fewer passengers, leading to significant inefficiencies in scenarios where passenger distribution is highly uneven.
* The algorithm **may struggle in scenarios where extreme outliers are present**. In cases where certain floors are significantly higher or lower than the majority, the algorithm might either overlook these outliers or suffer higher operational costs to include them, which can compromise overall efficiency and cost-effectiveness.

**Areas for Improvement and Future Research**

* **Investigate the use of other optimization algorithms** such as dynamic programming or branch and bound techniques to determine if they offer more effective solutions to elevator optimization problems than the current greedy approach.
* Check if it is possible to **combine the greedy method with other optimization strategies**, such as dynamic programming or branch and bound, to develop a more robust and comprehensive solution.
* **Integrate priority queues or hash maps within the greedy algorithm** to enhance operational efficiency and reduce the computational complexity of the optimizeGreedy() method, since arrays were used.
* **Refine the thresholds used in the adjustCostForSparseHighRanges() method** to better manage extreme outliers and prevent excessive operational costs.

# Conclusion

**Key Findings**

* The Greedy Algorithm successfully minimized walking distances for passengers by making informed decisions on which floors to stop, based on the number of passenger requests. We found that the algorithm was particularly effective in small to moderate-sized buildings with a limited number of stops allowed.
* Through extensive simulation and testing, we demonstrated that our algorithm can efficiently handle a variety of scenarios, from evenly distributed passenger loads to scenarios where passenger distribution is heavily skewed toward certain floors.
* The algorithm's strength lies in its simplicity and efficiency. Even if the algorithm has limitations for where extreme outliers are present (floors that are higher than 150), it is suitable enough for realistic and real-time applications in building management systems.

**Significance of the Results**

The success of the Greedy Algorithm in our project underscores the significant potential of algorithmic approaches in enhancing the efficiency of elevator systems where the number of stops is limited. By optimizing the number of stops and minimizing passenger walking distances, the algorithm can lead to reduced energy consumption and improved overall efficiency for the elevator. Moreover, the algorithm reduces the waiting time for the people going to the upper floors, since it reduces the number of stops on the lower floors.

**Future Directions and Practical Applications** **for Elevator Optimization Algorithm**

* **Multi-Origin Elevator Traffic**
  + Future research should explore optimization strategies for scenarios where passenger traffic is not limited to a single origin or destination floor, which would add a realistic layer to the elevator optimization simulation.
* **Public Transportation Efficiency**
  + Apply optimization principles such as this to public transit systems like subway stations or airports to streamline the movement of large crowds, reducing wait times and costs.
* **Residential and Commercial Applications**
  + Deploy the algorithm in residential high-rises and commercial buildings to manage elevator traffic during peak and off-peak hours.
* **Handling Priority Requests**
  + Integrate a priority handling mechanism to accommodate emergency stops and requests from individuals with special needs, ensuring they are serviced promptly while maintaining overall system efficiency.

In conclusion, our solution successfully implemented the Greedy Algorithm approach to optimize elevators where there are a limited number of stops, particularly focusing on minimizing the total walking distance for people going to other floors. This approach proved effective in handling the constraints of elevator stop optimization by making informed, locally optimal choices that accounted for the number of floors, passenger distribution each floor, and the sparseness of the floors. The implementation not only demonstrates the algorithm’s adaptability to varied scenarios but also sets a foundation for future enhancements that could create hybrid algorithms in solving the problem in better efficiency. This paves the way for smarter, more efficient building management systems, promising significant improvements in cost and operational efficiency.