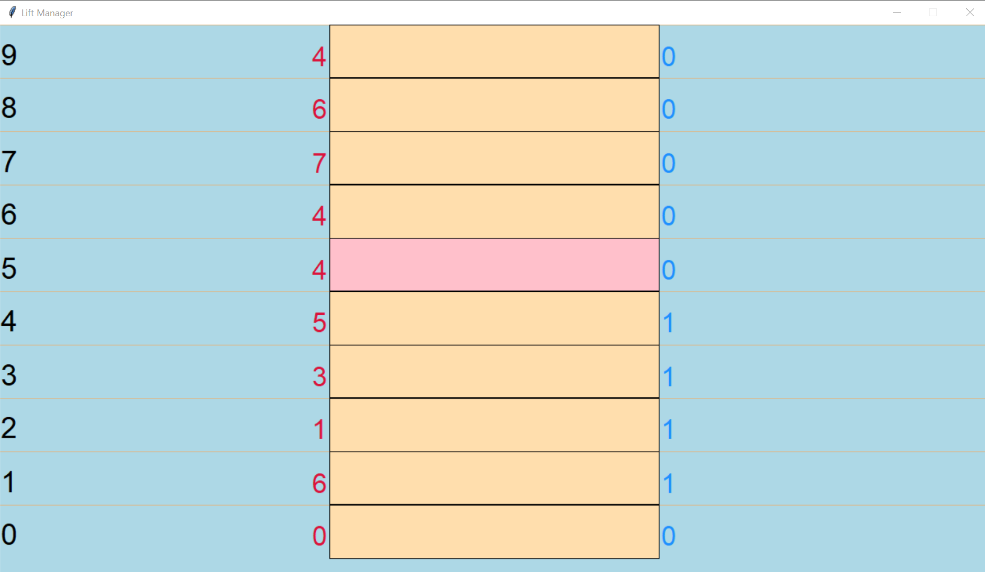
**Abstract**

The purpose of this report is to discuss and improve upon the systems relied upon to transport 325 million people every day. With this vast number of people relying on such an essential mode of transport, to simplify their daily lives, it seems strange that we rely on the same fundamental principles that have governed these systems since their inception in 1857 when the world's earliest modern 'safety elevator' went into service at the Haughwout Department Store in New York City, USA. It was installed by Elisha Otis and was powered by a steam engine. As an aside, the very same Otis Elevator Company claims that collectively its elevators are responsible today for carrying the equivalent of the world’s population every five days. This study aims to compare the ‘logic’ of those original systems with more current alternatives, to come up with a better method for moving individuals between floors, with the least ‘cost’ to each passenger as defined by the greatest reduction in wait time, whilst also attempting to minimise the total number of floors traversed in order to deliver all passengers to their desired destination.

Basic implementation

In order to get an idea for the inner workings of those more basic systems I began by implementing my own simplified version of the original mechanisms that operated early lifts with mechanical direction control. Such lifts required the cart to reach either the top or bottom floor in order to change the direction in which the lift motor operated.

To the left is a visual representation of the system in action which shows on the left-hand side the number of people waiting on each floor. The user can specify the maximum capacity as well as the number of floors in the building and how many passengers to generate before starting each run. The right-hand side displays the number of people delivered to each floor whilst the central column contains the lift shaft and cart shown in pink. This flashes green when collecting passengers and amber when delivering a person to a floor.



Improved system

From the improved lift manager I started by analysing some commonplace systems used elsewhere in industry and came across this article by Popular Mechanics detailing how modern algorithm perform a delicate balancing act to reduce the pain index felt by those waiting for the lift. This pain index attempts to quantify how people may respond when given the following options, one in which your elevator takes 10 seconds to arrive and then one minute to reach your destination, and another in which each portion takes 30 seconds. Many people find waiting so painful that they would prefer the first option, even though they reach their destination later. For each type of inconvenience, the system deliberates on how best to respond whether this be prioritising the speed at which a passenger is collected or alternatively reducing the overall time before they have reached their destination.

Through my research I have found that the most common original lift algorithm that still sees some use today is that of the SCAN algorithm detailed in an article by the Designing Buildings Wiki. This rather straight-forward logic is the same process by which some read/write heads on hard drive controllers determine the motion of the disks arm and head in servicing read and write request. The algorithm, however, is based originally on the way a lift works, where the elevator continues to travel in its current direction (up or down) until it is ‘empty’; it will stop only to let people off or to pick up people heading in the same direction. This manages to provide both a reasonably efficient means of delivering people whilst also traversing fewer unnecessary floors and the lift can change direction at any point and is no longer require reaching either end of the shaft. Thereby eliminating the few floors top and bottom that would ordinarily need to be covered before the lift was able to service people travelling in the opposite direction.

Such a system would have only marginal gains for buildings with a small number of floors, serving many passengers at once. There are however significant advantages to be gained if the lift is serving only a handful of passengers at a time in a building with many floors.

The example provided was one in which the building has 20 total ﬂoors. The lift is currently on the 7th ﬂoor with status as “up”. A person calls the lift on the 6th ﬂoor and they want to go “down” to the ground ﬂoor (ﬂoor 1) As a result of the mechanical system, the lift would have to travel to the 20th ﬂoor to change the direction and then return to the 6th ﬂoor to take the passenger who wants to go to the ground ﬂoor. This is quite inefﬁcient because the lift would have to travel 13 ﬂoors to the 20th, change the direction, and travel 20 ﬂoors to take the passenger to the ground ﬂoor. In the above scenario a system which were able to change direction at any time would be able to service the passing in only 6 moves, far quicker that the basic system previously described. To develop my own similar algorithm, I would need to employ the use of several different types of data structures to ensure that my system could deliver passengers more quickly while reducing wait times and with an overall greater efficiency for most of the simulations used to test and compare the systems.

By reducing the inefficiencies of the basic system and moving the lift only when required and in the direction most suitable there should be in most instances a significant improvement in both the satisfaction of people using the lift and well as the reduced cost in its operation. To this end my system when delivering a person to a floor with zero waiting floors is able to determine the most useful floor which it can travel to next. It does this by calculating the number of people waiting on floors either side of the lifts current position and instructing the system that the next floor to collect somebody should be that with the greater number of people waiting. If both floor either side are also empty, then this pattern repeats expanding outward until a suitable floor is found to which the lift can the travel.

Data structures and algorithms

The basic lift system utilises two sets of dictionaries, one containing the passenger who are yet to be collected and the other for those who have been delivered to the destination. Within both dictionaries are key value pairs where the key denotes the floor number and the value is a list containing every person either waiting or having arrived on that floor. Within the lift itself is a single list from which passenger are moved into and out of the lift when the depart and arrived at their desired floor. Each passenger is an object containing attributes such as their originating floor, destination, unique identifier and a counter for the number of floors the lift has travelled whilst they have been waiting to be collected. The lift is also an object containing the afore mentioned list of passengers, capacity, direction of travel, current position and number of moves totalled throughout the simulation. In addition to support the animation of the lift there is also a dictionary containing a reference to all the coloured blocks ‘tiles’ that make up each floor of the shaft through which it moves.

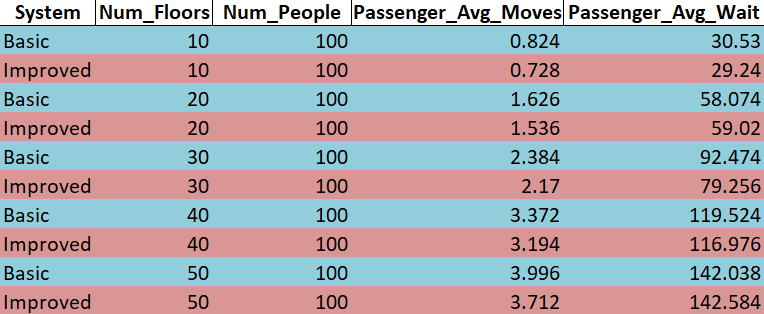
There is an extra layer of complexity added when it comes to designing the improved algorithm as this improved system also incorporates the use of sets as a means by which the next floor to travel to is determined based on how efficient or ‘useful’ this choice appears. Leading on from my previous explanation, for every floor the lift arrived on where there are no passengers to collect the lift runs function to determine every possible floor the lift could travel to if it continues moving in the same direction and compares this with a different set containing all the useful floors. Those which still have waiting passengers to be collected or alternatively a destination of any passenger currently remaining in the lift. The intersection of these two sets is taken and if any values exist in both sets then the function returns a value of True. If, however no intersection is found then this means that the lift has no further instructions to complete if it were to continue travelling in its current direction. As such, failure to find any intersection between the two sets of possible and useful floors results in the lift reversing direction immediately. The algorithm also attempts to implement a sort of queue albeit indirectly where for each of the passengers waiting on a given floor the lift services them in order of arrival at the lift such that when maximum capacity is reached those who entered the lift would be those who had been waiting the longest.

Below is a complete list of data structures and algorithms used to provide the two systems.

* Dictionaries
* Lists
* Sets
* Queues
* SCAN algorithm

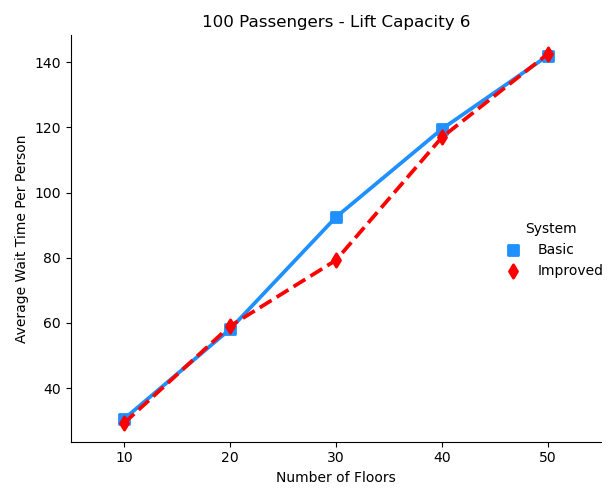
Analysis and comparison

In every scenario the lift begins moving away from the ground floor only after all the passengers have been generated and assigned to their respective floors.

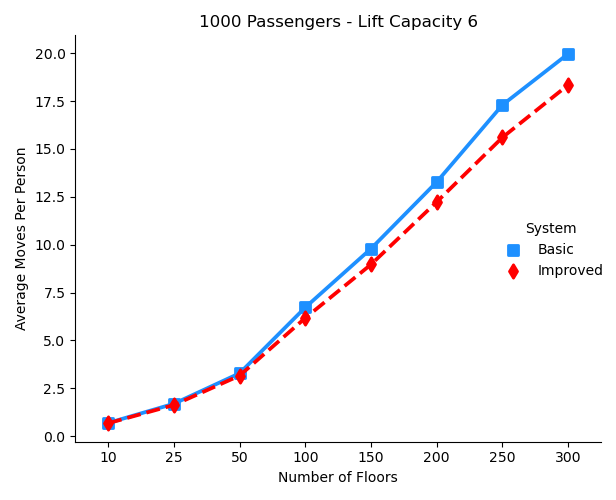


From the table of results and the graph below we can see that the improved algorithm performed better in all but one instance. This is inline with our prediction that given specific parameters the two algorithms were likely to perform equally as well or even in some instances like the one seen here the basic simulation would perform marginally better.

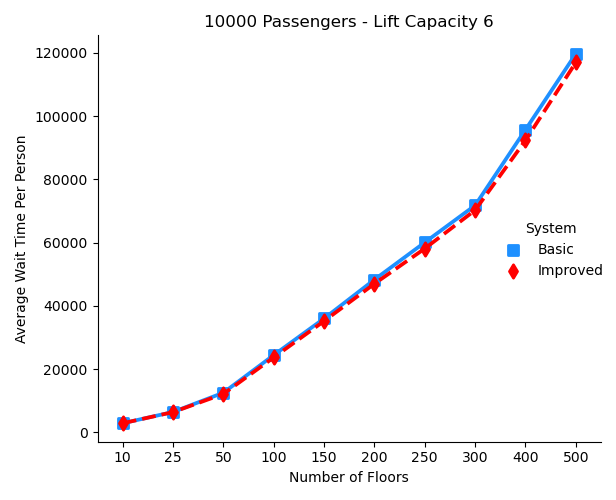
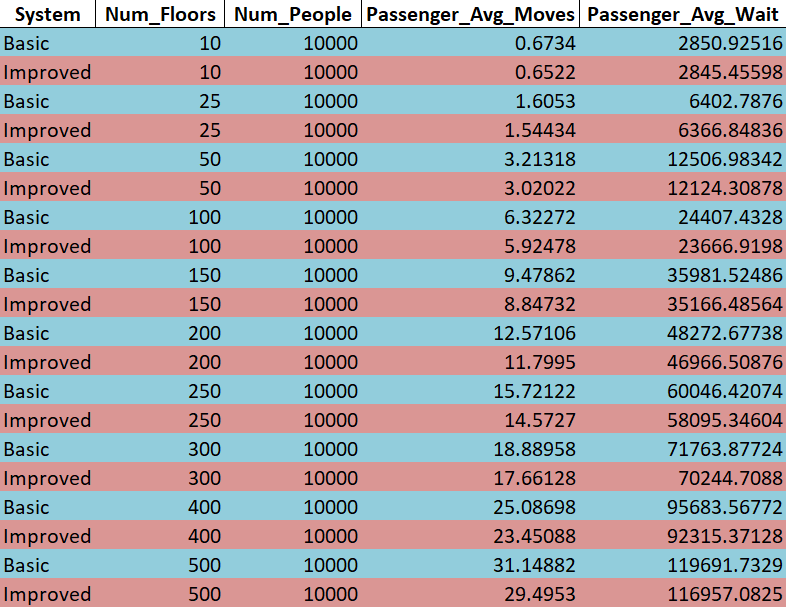
This graph allows us to clearly see at which particular intervals the improved algorithm largely outperforms the basic system and why in the majority of cases this would be the preferred option. For a building with 30 floors and 100 passengers the gap between the two lines representing the wait time comparison of the two algorithms is very clearly defined.



The curve below shows quite nicely how the improved algorithms performance tends further away from the greater number of average moves of the basic algorithm per person delivered and is a good summation of the efficiency sought by the improved system.

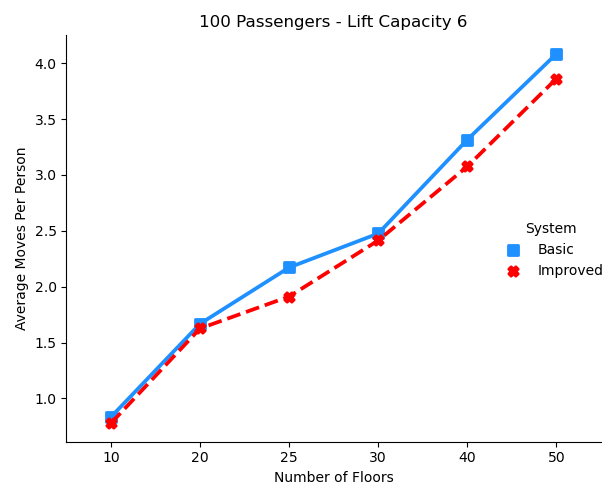


As discussed previously, because we are utilising the underlying fundamentals of the mechanical direction control, in that we are only picking up passengers travelling in the direction of the lift and only changing direction after having exhausted all possible moves. This inhibits us from making a significant reduction in the total number of moves needed to process every users request in smaller simulations, as the difference from the bottom to the top floor is not all the great, so we are likely to only improve marginally on the basic system in these cases. If instead we were to implement a destination dispatch system such as the one first used at the Marriott Hotel, New York then we have a greater change of seeing significant improvement, both in the rate at which people are collected as well as the time to arrival. Such a system assigns passengers to an elevator according to which floors they are going to, in an attempt to send each car to as few floors as possible. The person enters their desired floor at a control panel on each floor and is directed to which elevator to take. This required the elevators collective control logic knowing the user’s destination in advance of the carts moving anywhere and was not within the scope of the briefing for this project.



Even with these imposed limitations, if we look at a value closer to 120 taking a building with 100 floors for instance, shows that we are able to reduce the average wait time by almost 750 floors. Both the graph showing average passenger wait time and those comparing the total number of floors traversed by the lift are evidence to support the fact the improved algorithm is a more suitable alternative to the basic lift with mechanical direction control. Whilst this may be a long way off, with advancements in material science we are likely to soon overcome the limitations imposed by current day materials and if they ever come to fruition, algorithms such as the one detailed in this report may become of even more importance in running buildings like super skyscrapers and space elevators.

I wanted to ensure that my algorithm would perform well for both a relatively low number of people as well and a much greater number of people whilst varying the number of floors throughout my testing. As such I performed multiple simulations varying the number of floors with a fixed number of 10,000 people to be delivered each run. The results showed that even though the margin of improvement appears slim due to the large axis of the graph. Looking at the results showed a significant margin of improvement. Comparing the first and last pairs of results shows that when only using 10 floors the margin is rather small, with a reduction in waiting time of only five floors on average. However, if we were to increase the number of floors 500, we can find theoretical improvements of a reduction in wait time of nearly 3000 floors. This figure is currently unattainable due to the high limit of an elevator shaft with materials available right now. This is thought to be around 1640ft. Taking the Burj Khalifa as an example. The average height of a single story is 14ft giving us a theoretical limit of a lift capable of servicing at most 120 floors, before collapsing under the weight of its own cables snapping if it were any higher.



The graph to the left compares the average number of floors traversed by the lift per person to deliver everyone to their destination. This shows a similar trend to when we were comparing the average wait time per passenger. We can determine however that in this simulation whereas before there was a large difference around the 30-floor mark, in this example we can instead see two smaller regions either side of this value. This suggest that in measuring the number of moves made by the lift we may have a better idea of the overall efficiency of the new system rather than simply by comparing a passenger wait time. This would be useful in determining whether to prioritise collection time over time till delivery and the planning of future development of the algorithm.

Future scope

Whilst neither of my two algorithms service people in real time, if passengers were in fact generated on a random timed basis then there may be scope to include a parking algorithm to service floors with highest demand during different part of the day. This would work by recording the most commonly requested floors, and having the elevator waiting ‘parked’ ready to service any passengers who would be more likely to call the lift from these floors.

Whilst I ended up not utilising this system, I had experimented with including a priority queuing system to ensure that those who called the lift first were to be collected before others who arrived at the lift after. This algorithm would ensure that those waiting on a floor would be picked up in the order that they arrived. Whilst my current algorithm still includes this functionality these alternative algorithms also made provisions to ensure passengers waiting on other floors far from those the lift was currently serving would not be kept waiting for prolonged periods of time. It achieved this by selecting the passenger with the greatest distance to travel as the person to be delivered first and whilst making this journey would pick-up and drop-off any remaining passenger as and when necessary. This ensured that the lift was never operating in a single part of the building for too long and otherwise depriving certain passenger of service though a large number of short easily serviceable requests. This pattern would continue until one of the floors reached a counter of zero persons waiting at which point any journey bypassing this floor would become increasing inefficient as the lift would be unable to collect any additional passengers.

Using the link provided at the end of this report you will be able to view an example walk through of a priority queuing method compared to the basic algorithm. Through experimenting with this type of delivery mechanism I was able to deduce that to further increase the efficient of these algorithms the aim should be to average out the number of people waiting on each floor. This walkthrough details the movement of the lift and how the lift’s priority changes from delivering the greatest number of people before the waiting number of any floor reaches zero, to then servicing those whose journeys are the shortest.

Conclusion

In summary, from the evidence provided above it is fair to say that the improved system that I have devised it a better replacement for the basic system detailed in the project briefing. This very clearly meets the requirements for a ‘good’ elevator system in that it balances both users individual waiting time with those inconveniences that the user finds most annoying to provide a superior overall service to the mechanical direction control. Algorithms such as these are always being optimised as people’s needs change and new technologies emerge, and I am sure there are many other considerations that have been made to optimise more modern solutions, but for now this simple logic appears more than capable of serving most people’s everyday needs without cause for too much disruption.

Final words

Upon the conclusion of the of the project you will find additional resources as well as copies of all the source code and materials used to compile the report both at the link below as well as my GitHub repository. This includes the **videos, log** and **source-code** detailing the operation of the configurable system as well as a version of the code used to generate the data used along with any graphs. Of interest will be **lift.py** and **lift\_v96.py** which include the graphical operation of the system as well as a solely text-based representation, respectively.

If not viewed already, please take a look at the example walkthrough of the priority system.

OneDrive: <https://1drv.ms/u/s!Ar4UTJegDrU6n98QjYs4TlldT3ge-w?e=LiOzTY>

GitHub: <https://github.com/Will-Harris00/Lift-Controller>

References:

<https://www.connectionselevator.com/interesting-facts-you-might-not-know-about-elevators/>

<https://www.nytimes.com/2006/12/10/magazine/10section3b.t-1.html>

<https://theskydeck.com/how-tall-is-a-storey-in-feet/>

<https://www.21stcentech.com/materials-science-update-discovery-lead-mile-high-buildings/>

<https://www.newyorker.com/magazine/2008/04/21/up-and-then-down>

<https://www.guinnessworldrecords.com/world-records/first-building-to-have-a-safety-elevator-(lift)/>

<https://www.popularmechanics.com/technology/infrastructure/a20986/the-hidden-science-of-elevators/>

<https://www.designingbuildings.co.uk/wiki/The_mechanics_of_lift_routing>

Resources

**Source-code Graphical**

*from* tkinter *import* \*  
*from* tkinter *import* messagebox  
*import* random  
*import* time  
  
  
*class* VarEntry(Frame):  
 *def* \_\_init\_\_(self, *root*, *numFloors*=10, *numPeople*=50, *liftCapacity*=6, *numRepeats*=1, *delay*=0.35, *\*\*kw*):  
 super().\_\_init\_\_(\*\**kw*)  
 self.root = *root* self.error = StringVar()  
 self.error.set('If the number of repeats if greater that 1 then the animation will be turned off.')  
 self.errorLabel = Label(self.root, textvariable=self.error)  
 self.errorLabel.pack(side="bottom", fill='both')  
  
 # radio button selector to choose system logic.  
 self.systemLogic = BooleanVar()  
 self.systemLabel = Label(self.root, text="System logic: ")  
 self.systemLabel.pack(side="left", padx=4, pady=2)  
 self.inputSystem = Radiobutton(self.root, text="Improved", variable=self.systemLogic, value=*False*).pack(side="left")  
 self.inputSystem = Radiobutton(self.root, text="Basic", variable=self.systemLogic, value=*True*).pack(side="left")  
  
 self.floorsLabel = Label(self.root, text="Floors: ")  
 self.floorsLabel.pack(side="left")  
 self.inputNumFloors = Entry(self.root, textvariable="", width=5)  
 self.inputNumFloors.pack(side="left")  
  
 self.peopleLabel = Label(self.root, text="People: ")  
 self.peopleLabel.pack(side="left", padx=4, pady=2)  
 self.inputNumPeople = Entry(self.root, textvariable="", width=8)  
 self.inputNumPeople.pack(side="left", pady=2)  
  
 self.capacityLabel = Label(self.root, text="Lift Capacity: ")  
 self.capacityLabel.pack(side="left", padx=4, pady=2)  
 self.inputLiftCapacity = Entry(self.root, textvariable="", width=3)  
 self.inputLiftCapacity.pack(side="left", pady=2)  
  
 self.repeatsLabel = Label(self.root, text="Num Repeats: ")  
 self.repeatsLabel.pack(side="left", padx=4, pady=2)  
 self.inputNumRepeats = Entry(self.root, textvariable="", width=4)  
 self.inputNumRepeats.pack(side="left", pady=2)  
  
 self.delayLabel = Label(self.root, text="Delay Secs: ")  
 self.delayLabel.pack(side="left", padx=4, pady=2)  
 self.inputDelay = Entry(self.root, textvariable="", width=4)  
 self.inputDelay.pack(side="left", pady=2)  
 # assign default values  
 self.numFloors = *numFloors* self.numPeople = *numPeople* self.liftCapacity = *liftCapacity* self.numRepeats = *numRepeats* self.delay = *delay* self.inputNumFloors.insert(END, *numFloors*)  
 self.inputNumPeople.insert(END, *numPeople*)  
 self.inputLiftCapacity.insert(END, *liftCapacity*)  
 self.inputNumRepeats.insert(END, *numRepeats*)  
 self.inputDelay.insert(END, *delay*)  
  
 # bind left click on run bottom to validation of user input.  
 self.startBtn = Button(self.root, text="Run", fg="blue", width=5, command=self.validate)  
 self.startBtn.pack(side="right", padx=4, pady=2)  
  
 # alternatively binds enter/return key to validation of user input.  
 self.root.bind("<Return>", *lambda event*: self.validate())  
  
 # sets window title.  
 self.root.title("Data Entry Form")  
 # disables resizing of window, defaults to minimum size required to fit all elements.  
 self.root.resizable(*False*, *False*)  
 # specifies position on screen with default window sizing.  
 self.root.geometry("+250+250")  
 # defines exit protocol when clicking the red x button to close the program.  
 self.root.protocol("WM\_DELETE\_WINDOW", on\_continue)  
 # continuously check for any updates made to the window and canvas.  
 self.root.mainloop()  
  
 *def* validate(self):  
 *try*:  
 testFloors = int(self.inputNumFloors.get())  
 *try*:  
 testPeople = int(self.inputNumPeople.get())  
 *try*:  
 testCapacity = int(self.inputLiftCapacity.get())  
 *try*:  
 testRepeats = int(self.inputNumRepeats.get())  
 *try*:  
 testDelay = float(self.inputDelay.get())  
 *if* testFloors < 2:  
 self.error.set(  
 "The number of floors '" + str(  
 testFloors) + "' is out of range. Please choose a number greater that 1.")  
 print(  
 "The number of floors '" + str(  
 testFloors) + "' is out of range. Please choose a number greater than 1.")  
 *elif* testPeople < 1:  
 self.error.set(  
 "The number of people '" + str(  
 testPeople) + "' is out of range. Please choose a number greater than 0.")  
 print(  
 "The number of people '" + str(  
 testPeople) + "' is out of range. Please choose a number greater than 0.")  
 *elif* testCapacity < 1 *or* testCapacity > 16:  
 self.error.set(  
 "The capacity of the lift '" + str(  
 testCapacity) + "' is out of range. Please choose a number greater than 0.")  
 print(  
 "The capacity of the lift '" + str(  
 testCapacity) + "' is out of range. Please choose a number greater than 0.")  
 *elif* testRepeats < 1 *or* testRepeats > 1000:  
 self.error.set(  
 "The number of repeats '" + str(  
 testRepeats) + "' is out of range. Please choose a number between 1 and 1000.")  
 print(  
 "The number of repeats '" + str(  
 testRepeats) + "' is out of range. Please choose a number between 1 and 1000.")  
 *elif* testDelay < 0 *or* testDelay > 2:  
 self.error.set(  
 "The animation delay '" + str(  
 testDelay) + "' is out of range. Please choose a number between 0 and 2 seconds.")  
 print(  
 "The animation delay '" + str(  
 testDelay) + "' is out of range. Please choose a number between 0 and 2 seconds.")  
 *else*:  
 self.numFloors = testFloors  
 self.numPeople = testPeople  
 self.liftCapacity = testCapacity  
 self.numRepeats = testRepeats  
 self.delay = testDelay  
 self.systemLogic = self.systemLogic.get()  
 *if* self.systemLogic:  
 print("Basic system")  
 *else*:  
 print("Improved system")  
 print("Number of floors: " +str(self.numFloors))  
 print("Number of people: " +str(self.numPeople))  
 print("Lift capacity: " +str(self.liftCapacity))  
 print("Number of repetitions: " + str(self.numRepeats -1))  
 print("Number of floors: " +str(self.numFloors))  
 *if* self.numRepeats == 1:  
 self.animate = *True* print("Animation delay: " + str(self.delay))  
 *else*:  
 self.animate = *False* print("Animated: " + str(self.animate))  
 self.root.destroy()  
 *except*:  
 self.error.set("Please provide a valid input for animation delay.")  
 *except*:  
 self.error.set("Please provide a valid input for number of repeats.")  
 *except*:  
 self.error.set("Please provide a valid input for lift capacity.")  
 *except*:  
 self.error.set("Please provide a valid input for number of people.")  
 *except*:  
 self.error.set("Please provide a valid input for number of floors.")  
  
  
*class* Model():  
 *def* \_\_init\_\_(self, *master*):  
 self.departures = {}  
 self.arrivals = {}  
 self.master = *master* self.canvas = Canvas(self.master, width=*master*.winfo\_screenwidth()-20, height=*master*.winfo\_screenheight()-125, borderwidth=0,  
 highlightthickness=0,  
 bg="lightblue")  
 self.canvas.pack(fill="both", expand="true")  
  
 self.canvas.delete("nums")  
 self.canvas.delete("flrs")  
 self.canvas.delete("divs")  
 cellwidth = int(self.canvas.winfo\_reqwidth() / 3)  
 cellheight = int(round(self.canvas.winfo\_reqheight() / vars.numFloors))  
 *for* column *in* range(3):  
 current\_floor = vars.numFloors  
 *if* column == 1:  
 *for* row *in* range(vars.numFloors):  
 x1 = column \* cellwidth  
 y1 = row \* cellheight  
 x2 = x1 + cellwidth  
 y2 = y1 + cellheight  
 *if* row % 2 == 0:  
 y2 -= 1  
 line = self.canvas.create\_line(0, y1, self.canvas.winfo\_reqwidth(), y1,  
 fill="BurlyWood",  
 tags="divs")  
 # print("Line divider: " + str(line))  
 tile = self.canvas.create\_rectangle(x1, y1, x2, y2,  
 fill="NavajoWhite",  
 tags="flrs")  
 # print("Lift Tile: " + str(tile))  
 lift.tiles[current\_floor - 1] = tile  
 current\_floor -= 1  
 *elif* column *in* range(0, 3, 2):  
 floor\_num = 0  
 *for* row *in* range(vars.numFloors, 0, -1):  
 *if* column == 0:  
 y1 = (row \* cellheight) - 5  
 *if* row % 2 == 0:  
 y1 -= 1  
 y2 = y1 - (cellheight // 1.5)  
 num = self.canvas.create\_text(1, y2, anchor="nw", text=str(floor\_num), tags="flrs",  
 font=('Arial', -round(cellheight // 1.75)))  
 # print("Floor number: " + str(num))  
 *try*:  
 num\_waiting = len(waiting[floor\_num])  
 *except* KeyError:  
 num\_waiting = 0  
 departed\_num = self.canvas.create\_text(cellwidth - 3, y2 + 3, anchor="ne",  
 text=str(num\_waiting),  
 fill="Crimson",  
 font=('Arial', -round(cellheight // 2)))  
 self.departures[floor\_num] = departed\_num  
 # print("Depart number: " + str(departed\_num))  
 *elif* column == 2:  
 y1 = (row \* cellheight) - 5  
 *if* row % 2 == 0:  
 y1 -= 1  
 y2 = y1 - (cellheight // 1.5)  
 arrived\_num = self.canvas.create\_text((cellwidth \* column) + 3, y2 + 3, anchor="nw",  
 text="0",  
 fill="DodgerBlue",  
 font=('Arial', -round(cellheight // 2)))  
 self.arrivals[floor\_num] = arrived\_num  
 # print("Arrive number: " + str(arrived\_num))  
 floor\_num += 1  
 # print(lift.tiles)  
 # print(self.departures)  
 # print(self.arrivals)  
 self.master.title("Lift Manager")  
 self.master.protocol("WM\_DELETE\_WINDOW", on\_closing)  
 string\_geometry = str(*master*.winfo\_screenwidth()-20)+"x"+str(*master*.winfo\_screenheight()-75)+"+0+0"  
 print("Window geometry: " + string\_geometry)  
 self.master.geometry(string\_geometry)  
 self.master.resizable(*False*, *False*)  
 self.master.update()  
  
*class* improvedBuilding(object):  
 *def* \_\_init\_\_(self):  
 improvedBuilding.move(self)  
  
 *def* move(self):  
 first\_loop\_complete = *False* collect\_continue = *True  
 for* i *in* range(len(waiting)):  
 *if* len(waiting[i]) == 0:  
 *del* waiting[i]  
  
 *while* len(waiting) > 0 *or* len(lift.passengers) > 0:  
 print("\nThe lift is on floor " + str(lift.currentFloor))  
 *if* vars.animate == *True*:  
 tile = lift.tiles[lift.currentFloor]  
 model.canvas.itemconfigure(tile, fill="Pink")  
 model.canvas.update()  
 time.sleep(vars.delay)  
  
 # people are delivered before collecting others on the same floor  
 # to ensure optimal transportation as they must be travelling to  
 # a floor different from their origin, this frees up lift space.  
 deliver()  
  
 useful\_floors = set()  
  
 # print("\nPassengers travelling to these floors:")  
 *for* passenger *in* lift.passengers:  
 useful\_floors.add(passenger.destFlr)  
 # print(passenger.destFlr)  
 # print("\nPassengers waiting on these floors:")  
 *for* floor *in* waiting:  
 *if* len(waiting[floor]) != 0:  
 useful\_floors.add(floor)  
 # print("Floor " +str(floor))  
 # print("Num waiting " + str(len(waiting[floor])))  
  
 # print(useful\_floors)  
 possible\_flrs = set()  
 *for* i *in* range(0, numFloors - 1):  
 # print("Difference " + str(i))  
 *if* lift.direction == 1:  
 next\_floor = lift.currentFloor + i  
 *if* next\_floor > numFloors - 1:  
 *break* possible\_flrs.add(next\_floor)  
 *elif* lift.direction == -1:  
 next\_floor = lift.currentFloor - i  
 *if* next\_floor < 0:  
 *break* possible\_flrs.add(next\_floor)  
 # print(possible\_flrs)  
  
 *if* first\_loop\_complete:  
 *if* lift.currentFloor == numFloors - 1 *or* lift.currentFloor == 0:  
 lift.direction \*= -1  
 *elif not* ((bool(set(useful\_floors) & set(possible\_flrs)))):  
 lift.direction \*= -1  
 first\_loop\_complete = *True  
  
 if* collect\_continue:  
 collect()  
  
 *if* collect\_continue:  
 *if* len(waiting.keys()) == 1 :  
 *if* lift.currentFloor == list(waiting.keys())[0] *and* len(lift.passengers) == 0:  
 *for* person *in* waiting[lift.currentFloor][:]:  
 People.destination(person)  
 lift.passengers.append(person)  
 *if* vars.animate == *True*:  
 tile = lift.tiles[lift.currentFloor]  
 model.canvas.itemconfigure(tile, fill="ForestGreen")  
 model.canvas.update()  
 time.sleep(vars.delay / 5)  
 model.canvas.itemconfigure(tile, fill="Pink")  
 model.canvas.update()  
  
 waiting[lift.currentFloor].remove(person)  
 *if* len(waiting[lift.currentFloor]) == 0:  
 *del* waiting[lift.currentFloor]  
 # print(waiting)  
  
 *if* vars.animate == *True*:  
 # change the value of people waiting on that floor  
 departed\_num = model.departures[lift.currentFloor]  
 model.canvas.itemconfigure(departed\_num, text="0")  
 model.canvas.update()  
 time.sleep(vars.delay / 2)  
  
 print("\nPerson " + str(person.id) + " got in the lift at floor " + str(  
 lift.currentFloor))  
 print("There are " + str(len(lift.passengers)) + " passenger in the lift.")  
  
 *if* person.destFlr > lift.currentFloor:  
 lift.direction = 1  
 *else*:  
 lift.direction = -1  
 collect\_continue = *False  
  
 if* vars.animate == *True*:  
 tile = lift.tiles[lift.currentFloor]  
 model.canvas.itemconfigure(tile, fill="NavajoWhite")  
 model.canvas.update()  
 fin\_position = lift.currentFloor  
  
 lift.currentFloor += lift.direction  
 lift.floorsMoved += 1  
 *for* floor *in* waiting:  
 *for* person *in* waiting[floor]:  
 person.waitTime += 1  
  
 # need to remove the one extra move counted  
 # because the while loop runs to completion.  
 total\_wait\_time = 0  
 *for* floor *in* delivered:  
 *for* person *in* delivered[floor]:  
 total\_wait\_time += person.waitTime  
 print("\nThe lift travelled " + str(lift.floorsMoved) + " floors in total.")  
 print("The number of floors in the building was " + str(numFloors))  
 print("The number of people delivered is " + str(numPeople))  
 print("The average number of floors traversed to deliver each passenger is " + str(  
 lift.floorsMoved / numPeople))  
 print("The average wait-time per passenger is " + str(total\_wait\_time / numPeople))  
  
 *if* vars.animate == *True*:  
 # display the final floor the list ends on.  
 tile = lift.tiles[fin\_position]  
 model.canvas.itemconfigure(tile, fill="Pink")  
 model.canvas.update()  
 model.master.mainloop()  
  
  
*class* basicBuilding(object):  
 *def* \_\_init\_\_(self):  
 basicBuilding.move(self)  
  
  
 *def* move(self):  
 *while* len(waiting) > 0 *or* len(lift.passengers) > 0:  
 print("\nThe lift is on floor: " + str(lift.currentFloor))  
  
 *if* vars.animate == *True*:  
 tile = lift.tiles[lift.currentFloor]  
 model.canvas.itemconfigure(tile, fill="Pink")  
 model.canvas.update()  
 time.sleep(vars.delay)  
  
 # people are delivered before collecting others on the same floor  
 # to ensure optimal transportation as they must be travelling to  
 # a floor different from their origin, this frees up lift space.  
 deliver()  
 collect()  
  
 *if* vars.animate == *True*:  
 tile = lift.tiles[lift.currentFloor]  
 model.canvas.itemconfigure(tile, fill="NavajoWhite")  
 model.canvas.update()  
 fin\_position = lift.currentFloor  
  
 lift.currentFloor += lift.direction  
 lift.floorsMoved += 1  
 *for* floor *in* waiting:  
 *for* person *in* waiting[floor]:  
 person.waitTime += 1  
  
 *if* lift.currentFloor == numFloors - 1 *or* lift.currentFloor == 0:  
 lift.direction \*= -1  
  
 # need to remove the one extra move counted  
 # because the while loop runs to completion.  
 lift.floorsMoved -= 1  
 total\_wait\_time = 0  
 *for* floor *in* delivered:  
 *for* person *in* delivered[floor]:  
 total\_wait\_time += person.waitTime  
 print("\nThe lift travelled " + str(lift.floorsMoved) + " floors in total.")  
 print("The number of floors in the building was " + str(numFloors))  
 print("The number of people delivered is " + str(numPeople))  
 print("The average number of floors traversed to deliver each passenger is " + str(  
 lift.floorsMoved / numPeople))  
 print("The average wait-time per passenger is " + str(total\_wait\_time / numPeople))  
  
 *if* vars.animate == *True*:  
 # display the final floor the list ends on.  
 tile = lift.tiles[fin\_position]  
 model.canvas.itemconfigure(tile, fill="Pink")  
 model.canvas.update()  
 model.master.mainloop()  
  
*def* collect():  
 *try*:  
 *if* len(waiting[lift.currentFloor]) == 0:  
 *del* waiting[lift.currentFloor]  
 *for* person *in* waiting[lift.currentFloor][:]:  
 # print("Person " + str(person.id) + " is travelling in direction: " + str(person.direction) + " the lift direction is: " + str(lift.direction))  
 # adds waiting passengers to the lift if travelling in the direction of the lift.  
 *if* person.direction == lift.direction:  
 People.destination(person)  
 *if* len(lift.passengers) < lift.capacity:  
 lift.passengers.append(person)  
  
 *if* vars.animate == *True*:  
 tile = lift.tiles[lift.currentFloor]  
 model.canvas.itemconfigure(tile, fill="ForestGreen")  
 model.canvas.update()  
 time.sleep(vars.delay/5)  
 model.canvas.itemconfigure(tile, fill="Pink")  
 model.canvas.update()  
  
 waiting[lift.currentFloor].remove(person)  
  
 *if* vars.animate == *True*:  
 # change the value of people waiting on that floor  
 departed\_num = model.departures[lift.currentFloor]  
 model.canvas.itemconfigure(departed\_num, text=str(len(waiting[lift.currentFloor])))  
 model.canvas.update()  
 time.sleep(vars.delay/2)  
  
 *if* len(waiting[lift.currentFloor]) == 0:  
 *del* waiting[lift.currentFloor]  
 print("\nPerson " + str(person.id) + " got in the lift at floor " + str(lift.currentFloor))  
 print("There are " + str(len(lift.passengers)) + " passenger in the lift.")  
 # saves searching through the remaining passengers if the lift is already full.  
 *else*:  
 *break  
 except*:  
 *try*:  
 *if* len(waiting[lift.currentFloor]) == 0:  
 *del* waiting[lift.currentFloor]  
 *except*:  
 *pass  
 pass  
  
  
def* deliver():  
 *for* person *in* lift.passengers[:]:  
 *if* person.destFlr == lift.currentFloor:  
 delivered[person.destFlr].append(person)  
  
 *if* vars.animate == *True*:  
 tile = lift.tiles[lift.currentFloor]  
 model.canvas.itemconfigure(tile, fill="Orange")  
 model.canvas.update()  
 time.sleep(vars.delay/5)  
 model.canvas.itemconfigure(tile, fill="Pink")  
 model.canvas.update()  
  
 lift.passengers.remove(person)  
  
 *if* vars.animate == *True*:  
 # change the value of people having arrived on that floor.  
 arrive\_num = model.arrivals[lift.currentFloor]  
 model.canvas.itemconfigure(arrive\_num, text=str(len(delivered[lift.currentFloor])))  
 model.canvas.update()  
 time.sleep(vars.delay/2)  
  
 print("Person " + str(person.id) + " exited the lift on floor " + str(person.destFlr))  
 print("There are " + str(len(lift.passengers)) + " passengers in the lift.")  
  
  
*class* Lift(object):  
 *def* \_\_init\_\_(self):  
 self.capacity = vars.liftCapacity  
 self.currentFloor = 0  
 self.direction = 1  
 self.floorsMoved = 0  
 self.passengers = []  
 self.tiles = {}  
  
  
*class* People(object):  
 *def* \_\_init\_\_(self, *topFloor*, *id*):  
 self.id = *id* self.waitTime = 0  
 self.originFlr = random.randint(0, *topFloor*)  
  
 # if the person is on the top floor then they must travel down.  
 *if* self.originFlr == *topFloor*:  
 self.direction = -1  
 # if the person is on the bottom floor then they must be travelling up.  
 *elif* self.originFlr == 0:  
 self.direction = 1  
 # randomly selects whether the passenger is travelling up or down  
 *else*:  
 self.direction = random.choice([-1, 1])  
 # print("\n"+str(self.originFlr))  
 # print(self.direction)  
 # print(self.destFlr)  
  
 *def* destination(self):  
 # A destination floor is generated based on whether the passenger is  
 # travelling up or down where only an applicable floors will be chosen.  
 *if* self.direction == 1:  
 # direction 1 shows the passenger wishes to travel to a higher floor  
 selection = list(range(self.originFlr + 1, numFloors))  
 self.direction = 1  
 *else*:  
 # direction -1 show the passenger wishes to travel to a lower floor  
 selection = list(range(0, self.originFlr))  
 self.destFlr = random.choice(selection)  
  
  
*def* on\_continue():  
 *if* messagebox.askokcancel("Run animation", "Do you want to continue with default values?"):  
 print("\nRunning simulation with default values.\n")  
 root.destroy()  
  
  
*def* on\_closing():  
 *if* messagebox.askokcancel("Exit program", "Do you want to quit?"):  
 master.destroy()  
  
*if* \_\_name\_\_ == "\_\_main\_\_":  
 # root is the entry window that validates the user input.  
 root = Tk()  
 # VarEntry is the class containing the user's input  
 vars = VarEntry(root)  
  
 *if* vars.animate == *True*:  
 # master is the animation window  
 master = Tk()  
  
 *while* vars.numRepeats > 0:  
 print("\nSimulation number: " + str(vars.numRepeats))  
 # creates the lift object and add the index of tiles to a dictionary  
 lift = Lift()  
  
 waiting = {}  
 delivered = {}  
 numFloors = vars.numFloors  
 numPeople = vars.numPeople  
  
 # initialises the dictionaries  
 *for* i *in* range(numFloors):  
 waiting[i] = []  
 delivered[i] = []  
  
 *for* id *in* range(0, numPeople):  
 person = (People(numFloors - 1, id))  
 waiting[person.originFlr].append(person)  
 print(waiting)  
  
 *if* vars.animate == *True*:  
 # Model is the class containing the building objects  
 model = Model(master)  
  
 *if* vars.systemLogic:  
 stats = basicBuilding()  
 *else*:  
 stats = improvedBuilding()  
  
 vars.numRepeats -= 1

**Source-code Logical**

*import* random  
*import* numpy *as* np  
*import* matplotlib.pyplot *as* plt  
*import* pandas *as* pd  
*from* openpyxl *import* load\_workbook  
*import* seaborn *as* sns  
# this is a copy of the finished improved algorithm from version 63  
# this will be updated to include any code required for the statistical analysis  
  
  
# ensure data frame is not truncated  
pd.set\_option('display.max\_columns', *None*)  
pd.set\_option('display.max\_rows', *None*)  
  
*class* basicBuilding(object):  
 *def* \_\_init\_\_(self):  
 waiting = {}  
 delivered = {}  
  
 # initialises the dictionaries  
 *for* i *in* range(numFloors):  
 waiting[i] = []  
 delivered[i] = []  
  
 *for* id *in* range(0, numPeople):  
 person = (People(numFloors - 1, id))  
 waiting[person.originFlr].append(person)  
 # print(waiting)  
 basicBuilding.move(self)  
  
  
 *def* move(self):  
 *while* len(waiting) > 0 *or* len(lift.passengers) > 0:  
 # print("\nThe lift is on floor: " + str(lift.currentFloor))  
  
 # people are delivered before collecting others on the same floor  
 # to ensure optimal transportation as they must be travelling to  
 # a floor different from their origin, this frees up lift space.  
 deliver()  
 collect()  
  
 lift.currentFloor += lift.direction  
 lift.floorsMoved += 1  
 *for* floor *in* waiting:  
 *for* person *in* waiting[floor]:  
 person.waitTime += 1  
  
 *if* lift.currentFloor == numFloors - 1 *or* lift.currentFloor == 0:  
 lift.direction \*= -1  
  
 # need to remove the one extra move counted  
 # because the while loop runs to completion.  
 lift.floorsMoved -= 1  
 self.total\_wait\_time = 0  
 *for* floor *in* delivered:  
 *for* person *in* delivered[floor]:  
 self.total\_wait\_time += person.waitTime  
  
 print("\nThe lift travelled " + str(lift.floorsMoved) + " floors in total.")  
 print("The number of floors in the building was " + str(numFloors))  
 print("The number of people delivered is " + str(numPeople))  
 print("The average number of floors traversed to deliver each passenger is " + str(  
 lift.floorsMoved / numPeople))  
 print("The average wait-time per passenger is " + str(self.total\_wait\_time / numPeople) + "\n")  
  
  
*class* improvedBuilding(object):  
 *def* \_\_init\_\_(self):  
 improvedBuilding.move(self)  
  
 *def* move(self):  
 first\_loop\_complete = *False* collect\_continue = *True  
 for* i *in* range(len(waiting)):  
 *if* len(waiting[i]) == 0:  
 *del* waiting[i]  
  
 *while* len(waiting) > 0 *or* len(lift.passengers) > 0:  
 # print("\nThe lift is on floor " + str(lift.currentFloor))  
  
 # people are delivered before collecting others on the same floor  
 # to ensure optimal transportation as they must be travelling to  
 # a floor different from their origin, this frees up lift space.  
 deliver()  
  
 useful\_floors = set()  
  
 # print("\nPassengers travelling to these floors:")  
 *for* passenger *in* lift.passengers:  
 useful\_floors.add(passenger.destFlr)  
 # print(passenger.destFlr)  
 # print("\nPassengers waiting on these floors:")  
 *for* floor *in* waiting:  
 *if* len(waiting[floor]) != 0:  
 useful\_floors.add(floor)  
 # print("Floor " +str(floor))  
 # print("Num waiting " + str(len(waiting[floor])))  
  
 # print(useful\_floors)  
 possible\_flrs = set()  
 *for* i *in* range(0, numFloors - 1):  
 # print("Difference " + str(i))  
 *if* lift.direction == 1:  
 next\_floor = lift.currentFloor + i  
 *if* next\_floor > numFloors - 1:  
 *break* possible\_flrs.add(next\_floor)  
 *elif* lift.direction == -1:  
 next\_floor = lift.currentFloor - i  
 *if* next\_floor < 0:  
 *break* possible\_flrs.add(next\_floor)  
 # print(possible\_flrs)  
  
 *if* first\_loop\_complete:  
 *if* lift.currentFloor == numFloors - 1 *or* lift.currentFloor == 0:  
 lift.direction \*= -1  
 *elif not* ((bool(set(useful\_floors) & set(possible\_flrs)))):  
 lift.direction \*= -1  
 first\_loop\_complete = *True  
  
 if* collect\_continue:  
 collect()  
  
 *if* collect\_continue:  
 *if* len(waiting.keys()) == 1 :  
 *if* lift.currentFloor == list(waiting.keys())[0] *and* len(  
 lift.passengers) == 0:  
 *for* person *in* waiting[lift.currentFloor][:]:  
 People.destination(person)  
 lift.passengers.append(person)  
  
 waiting[lift.currentFloor].remove(person)  
 *if* len(waiting[lift.currentFloor]) == 0:  
 *del* waiting[lift.currentFloor]  
 # print(waiting)  
  
 # print("\nPerson " + str(person.id) + " got in the lift at floor " + str(lift.currentFloor))  
 # print("There are " + str(len(lift.passengers)) + " passenger in the lift.")  
  
 *if* person.destFlr > lift.currentFloor:  
 lift.direction = 1  
 *else*:  
 lift.direction = -1  
 collect\_continue = *False* lift.currentFloor += lift.direction  
 lift.floorsMoved += 1  
 *for* floor *in* waiting:  
 *for* person *in* waiting[floor]:  
 person.waitTime += 1  
  
 # need to remove the one extra move counted  
 # because the while loop runs to completion.  
 self.total\_wait\_time = 0  
 *for* floor *in* delivered:  
 *for* person *in* delivered[floor]:  
 self.total\_wait\_time += person.waitTime  
  
 print("\nThe lift travelled " + str(lift.floorsMoved) + " floors in total.")  
 print("The number of floors in the building was " + str(numFloors))  
 print("The number of people delivered is " + str(numPeople))  
 print("The average number of floors traversed to deliver each passenger is " + str(  
 lift.floorsMoved / numPeople))  
 print("The average wait-time per passenger is " + str(self.total\_wait\_time / numPeople) + "\n")  
  
  
*def* collect():  
 *try*:  
 *if* len(waiting[lift.currentFloor]) == 0:  
 *del* waiting[lift.currentFloor]  
 *for* person *in* waiting[lift.currentFloor][:]:  
 # print("Person " + str(person.id) + " is travelling in direction: " + str(person.direction) + " the lift direction is: " + str(lift.direction))  
 # adds waiting passengers to the lift if travelling in the direction of the lift.  
 *if* person.direction == lift.direction:  
 People.destination(person)  
 *if* len(lift.passengers) < lift.capacity:  
 lift.passengers.append(person)  
  
 waiting[lift.currentFloor].remove(person)  
  
 *if* len(waiting[lift.currentFloor]) == 0:  
 *del* waiting[lift.currentFloor]  
 # print("\nPerson " + str(person.id) + " got in the lift at floor " + str(lift.currentFloor))  
 # print("There are " + str(len(lift.passengers)) + " passenger in the lift.")  
 # saves searching through the remaining passengers if the lift is already full.  
 *else*:  
 *break  
 except*:  
 *try*:  
 *if* len(waiting[lift.currentFloor]) == 0:  
 *del* waiting[lift.currentFloor]  
 *except*:  
 *pass  
 pass  
  
def* deliver():  
 *for* person *in* lift.passengers[:]:  
 *if* person.destFlr == lift.currentFloor:  
 delivered[person.destFlr].append(person)  
  
 lift.passengers.remove(person)  
  
 # print("Person " + str(person.id) + " exited the lift on floor " + str(person.destFlr))  
 # print("There are " + str(len(lift.passengers)) + " passengers in the lift.")  
  
  
*class* Lift(object):  
 *def* \_\_init\_\_(self):  
 self.capacity = capacity  
 self.currentFloor = 0  
 self.direction = 1  
 self.floorsMoved = 0  
 self.passengers = []  
 self.tiles = {}  
  
  
*class* People(object):  
 *def* \_\_init\_\_(self, *topFloor*, *id*):  
 self.id = *id* self.waitTime = 0  
 self.originFlr = random.randint(0, *topFloor*)  
  
 # if the person is on the top floor then they must travel down.  
 *if* self.originFlr == *topFloor*:  
 self.direction = -1  
 # if the person is on the bottom floor then they must be travelling up.  
 *elif* self.originFlr == 0:  
 self.direction = 1  
 # randomly selects whether the passenger is travelling up or down  
 *else*:  
 self.direction = random.choice([-1, 1])  
 # print("\n"+str(self.originFlr))  
 # print(self.direction)  
 # print(self.destFlr)  
  
 *def* destination(self):  
 # A destination floor is generated based on whether the passenger is  
 # travelling up or down where only an applicable floors will be chosen.  
 *if* self.direction == 1:  
 # direction 1 shows the passenger wishes to travel to a higher floor  
 selection = list(range(self.originFlr + 1, numFloors))  
 self.direction = 1  
 *else*:  
 # direction -1 show the passenger wishes to travel to a lower floor  
 selection = list(range(0, self.originFlr))  
 self.destFlr = random.choice(selection)  
  
  
*if* \_\_name\_\_ == "\_\_main\_\_":  
 # Max lift capacity  
 max\_capacity = [6]  
 # system selection  
 systems = ['Basic', 'Improved']  
 # Number of floors  
 num\_floors\_sims = [10, 20, 30, 40, 50]  
 # Number of people  
 num\_people\_sims = [100]  
 j = 0  
 *for* capacity *in* max\_capacity:  
 excel\_file = "LiftCapacity" + str(capacity) + ".xlsx"  
 df = pd.DataFrame(columns=["System", "Num\_Floors", "Num\_People",  
 "Passenger\_Avg\_Moves",  
 "Passenger\_Avg\_Wait"])  
 *for* system *in* systems:  
 *for* numFloors *in* num\_floors\_sims:  
 *for* numPeople *in* num\_people\_sims:  
 # runs every combination five times and takes the average  
 combined\_total\_wait\_time = 0  
 combined\_total\_lift\_moves = 0  
 *for* k *in* range(5):  
 print("\nSimulation number: " + str(j + 1) + ", Run: " + str(k + 1))  
 print("Using the '" + system + "' system" )  
 print("The number of floors is: " + str(numFloors))  
 print("The number of people is: " + str(numPeople))  
 print("The lift capacity is: " + str(capacity))  
 # creates the lift object  
 lift = Lift()  
  
 waiting = {}  
 delivered = {}  
  
 # initialises the dictionaries  
 *for* i *in* range(numFloors):  
 waiting[i] = []  
 delivered[i] = []  
  
 *for* id *in* range(0, numPeople):  
 person = (People(numFloors - 1, id))  
 waiting[person.originFlr].append(person)  
 # print(waiting)  
  
 *if* system == "Basic":  
 stats = basicBuilding()  
 *else*:  
 stats = improvedBuilding()  
  
 combined\_total\_lift\_moves += (lift.floorsMoved / numPeople)  
 combined\_total\_wait\_time += (stats.total\_wait\_time / numPeople)  
  
 avg\_total\_lift\_moves = combined\_total\_lift\_moves / 5  
 avg\_total\_wait\_time = combined\_total\_wait\_time / 5  
 print(avg\_total\_lift\_moves)  
 print(avg\_total\_wait\_time)  
 df.loc[j] = [system, numFloors, numPeople,  
 avg\_total\_lift\_moves,  
 avg\_total\_wait\_time]  
 j+=1  
 print(df)  
 df.to\_excel(excel\_file, sheet\_name='Sheet\_name\_1')  
 # Use the 'hue' argument to provide a factor variable  
 sns.catplot(data=df, x="Num\_Floors", y="Passenger\_Avg\_Wait",  
 hue='System', kind='point',  
 legend='full',  
 palette={'Basic': 'dodgerblue', 'Improved': 'red'},  
 linestyles=["-", "--"], markers=["s", "d"])  
 plt.title(str(num\_people\_sims[0]) + " Passengers - Lift Capacity " + str(capacity))  
 plt.xlabel('Number of Floors')  
 plt.ylabel('Average Wait Time Per Person')  
 plt.show()

**Log**

08/01/20

Evaluated the guidance concerning our Data Structure and Algorithms Coursework and started planning some of the various features and functionality that would eventually make it into the final project.

Different Elevators Having Different Speeds

Different Elevators having overlap between certain floors

Different elevators having varying capacities

Service elevators being used for fire evacuation of a building

Start of the day having several elevators at the ground floor

Priority of different floors

Example of a hotel with bottom floor reception

Living quarters in-between

Penultimate floor has a gym and sauna

Top floor has a swimming pool

Questions you should be asking yourselves (not limited to these)

• What do you want to minimise?

• Wait-time of people? (recommended)

• Do you have another idea? Ask me or the TAs

• What is the max capacity of the lift and how do you control for this?

• How is the lift called?

• Two-button system: up and down (recommended)

• What if the building has multiple lifts?

• Start with 1 lift!

Baseline is a lift that can only change direction when it reaches either the top or bottom floor of the building. Start with the case where you have a building with only 10 floors.

Be sure not to keep people in a permanent status of waiting.

Graph for evaluation of efficiency using number of floors of movement to serve one user and the number of floors this user is being moved.

Random generators function to simulate people travelling within the building.

Cumulative weight of all users in the elevator, stops the lift from moving if overcapacity.

15/01/20

Started working on a simple user interface using PyQt which would allow the user to input the number of lifts as well as the range of floors that each individual lift has access to. This would later link to a tkinter animation showing the lift moving up and down between floors. This would later be extended to include random passengers moving between floors to test the efficiency of the system.

23/01/20

Designed a simple animation with a square moving up and down representing the lift movement.

Later I will add floor numbers and allow the lift to stop at each floor to allow passenger on and off.

The first version of my lift animation generates a lift shaft which stretches from the top of the canvas to the bottom. Inside there is a lift which moves smoothly up and down and changes direction whenever it reaches either end of the lift shaft. Currently much of the positioning and sizing elements are generated using numerical data this will later need to be changed such that the animation can adapt dynamically to the size of the canvas, number of lifts and number of floors.

25/01/20

After having experimented for a while with the various animation styles I came across a method to split the canvas into multiple cell forming a grid structure. This would allow me to change the colour of each cell individually representing the movement of a lift between floors. I started with generic code used to generate a checkers board of square cells which change colour if clicked by the user. The script was modified to allow it to be resized and updated with a different number of column and rows which would later represent the number of floors and number of lift shaft within the building. From here I was able to create alternating columns the first of which contains the floor number and the second would be used for the lift animation.

Now that I had two separate columns for each lift, I started created a function that would fill in every other column with the floor numbers. This did not end up working as intended because instead of filling in the blank column I had left for the floor numbers, the script instead generated a second canvas which it was overlaying on-top of the first set of columns on the main canvas. I later realised that I would be better off rather than using a label function to format the cell to instead use a rectangular shape and fill this with block text. From here I needed to invert the sequence in which the canvas filled in each of the floor numbers as the top floor was originally being labelled as floor zero and vice-verse.

During the process of filling all the cells I am recording a dictionary of the cells representing the moving lift and assigning each a number. This will allow me to quickly change the properties any given cell based on its location within the grid. In order to allow for the resizing of the canvas window each cell representing the lift is assigned the tag ‘lfts’ and any floor number ‘flrs’ this allows the canvas to be wiped and regenerated much faster than replacing every cell individually whenever the end-user resized the window. The next problem to solver was how to allow the text to dynamically resize relative to the size of the cells. This was done by finding the height of each cell in pixels and using floor division to find an integer number of pixels such that the height of the text was two thirds the height of the cell and would shrink and grow with the resizing of the window. For some unknown reason even though I was using floor division which should have returned an integer value I ended up also needing to use a rounding function to remove the trailing zero decimal. In later versions I will revisit this to increase the systems efficiency by reducing the number of operations. There is also a conditional statement for renaming the 0th floor to G for ground floor. Later I will include various basement levels for negative floor numbers or even allow the user to rename a cell by clicking on the text. This would be useful in various building layouts such as a rooftop pool or gym in a hotel or multiple carpark levels in a shopping mall. In terms of aesthetics I would like to have different sized columns. Currently the canvas is split in half which means there is a lot of wasted space in the cell containing the floor numbers. I would prefer to have these cells be smaller compare to those animating the lifts movement.

26/01/2020

I started today by creating two new functions based on the standard case of having a single lift serve a building of ten floors. The first of these functions would generate a list of random numbers where adjacent pairs would represent the starting floor of a passenger and the second the final destination. From here this list would be passed into the move function which would colour the starting position of the lift and switch the colour of adjected cells such that the lift appeared as though it were moving up and down between floors. Because I was using a for I in range loop to edit the colour of the cells the function would fail whenever the first number in the pair was lower than the second. This is due to the need to specify a negative step sequence when finding a decreasing range (8,2,-1). I would therefore incorporate an if else statements to determine whether the starting floor was greater than or less than the second. Later on I would also need to ensure that the random number generated would ignore any pairs generated where the numbers were identical but during testing this happened so infrequently that I decided to continue development and leave this till the end. Changed the naming scheme for keys within the dictionary to (Lift\_num, Floor) instead of the originally which was (row, columns) this made referencing each cell numerically much easier as for whatever reason the developers of tkinter decided to refer to cell by a single numerical value starting from the top left corner and counting down to the bottom of each column before moving to the next column this count also starts from one instead of zero unlike every other python module.

This table shows how I was originally creating the keys for my dictionary store.

|  |  |  |  |
| --- | --- | --- | --- |
| (0, 0) | (0, 1) | (0, 2) | (0, 3) |
| (1, 0) | (1, 1) | (1, 2) | (1, 3) |
| (2, 0) | (2, 1) | (2, 2) | (2, 3) |
| (3, 0) | (3, 1) | (3, 2) | (3, 3) |

|  |  |  |
| --- | --- | --- |
| 1 | 5 | 9 |
| 2 | 6 | 10 |
| 3 | 7 | 11 |
| 4 | 8 | 12 |

The table below represents the value stored by each of the dictionary keys above. These values are the numerical id of each individual cell which Tkinter uses to apply a function on the selected cell such as changing its colour. This shown the order in which each cell is assigned a unique id number.

By changing the keys of the dictionary to look like the ones below I am now able to move the lift up and down by looking for the lift number and the floor numbers that I wish to move it between. This greatly reduces the number of calculations needed to find the cell id number thereby increasing the efficiency of the program. Of course, all these key values will be numerical in future versions further increasing the efficiency of animating the movement of the lift in the final project.

|  |  |  |  |
| --- | --- | --- | --- |
| (1, *top\_floor*) | (2, *top\_floor*) | (3, *top\_floor*) | (4, *top\_floor*) |
| (1, 2) | (2, 2) | (3, 2) | (4, 2) |
| (1, 1) | (2, 1) | (3, 1) | (4, 1) |
| (1, *ground*) | (2, *ground*) | (3, *ground*) | (4, *ground*) |

Again I needed to invert the numbering scheme for the floors in the dictionary key tuple otherwise I would need an additional operation later on to find the difference between the max number of floors and the starting floor of the passenger in order to find the cell id number by subtracting the difference from the top floor number. By setting up the dictionary correctly before performing other functions with the dictionary I would later be able to easily reference each cell without involving additional math.

I now faced a problem with indexing as the top floor would be referred to by (1, 9) in a ten floor building counting from G or zero upwards therefore I needed to limit my range of random numbers by decrementing the number of floors by one. I decided to store the numerical value of the top floor in a separate variable as this would be referenced in multiple functions later to ensure the lift did not travel higher than the top story of the building. Doing so allows me to cut down on the number of operation as I would no longer need to subtract 1 from the ‘num\_floors’ variable each time I generated a random list or wanted to lookup the cell id of the top floor in the ‘lift\_pos’ dictionary.

I forgot to regenerate a new set of random numbers whenever a pair of matching numbers appeared in the list. This was a quick fix but should have been something I was looking out for.

I would now split the single function I had responsible for highlighting the starting floor and outputting the range of floors between the starting and end floor into two separate functions. This would allow me to highlight and unhighlight adjacent cells moving up and down the column between the passenger’s floors giving the appearance that the lift was moving up and down. I could call the second function multiple times until the lift had come to a stop. This would also make future development and improvements to the algorithm much simpler as I only required the cell ids for this function to work and could use separate algorithms for choosing which order passenger are served.

29/01/20

Found an issue with resizing the window which lead to the random lists being regenerated as this function would be called as a result of the root canvas being updated. Therefore, I needed to make this a stand-alone function. By doing so I would have an easier time later in the development process when trying to implement threading to run the lifts simultaneously as opposed to my originally idea of switching between lifts when completing each successive operation. The result of this was that the starting position of the lift flickered around whenever the window was resized.

01/02/20

I recently switched IDE’s from JetBrains’ PyCharm to Microsoft’s Visual Studio Code. Upon running my program, I realised that whenever closing out of the TKinter window by clicking the red cross button VS Code would attempt to forcibly keep the program running. To solve this issue, I implemented a function responsible for creating a popup box wherein the user could choose to quit the program or to return to the main window. Upon clicking quit the function calls window.destroy() which safely closes out of the program and stops VS Code from continuously reopening the window. Since receiving the official briefing on the project, I have changed the lift mechanism to always begin at the ground floor and run the generated sequence from there. Resizing the windows now saves the state of the program including the current position of the lift. I have updated the random generator to include the direction that the passenger wishes to travel as well as the floor that the passenger is starting on so that when the lift reaches them the destination floor is then determined.

02/02/20

Now that I had most of the UI complete, I decided to create a separate module for the algorithm responsible for running the lift. For this I would to include functions to deal with:

* Deciding which passenger should be picked-up and in which order.
* Counting the number of floors travelled for the entire sequence
* Comparison of the more efficient system with the naive system stated in the brief
* Queuing system to ensure no passenger is left waiting for a prolonged period
* Passenger exceeding the maximum capacity of the lift
* Calculating average wait time per passenger to use in the analysis of both systems

Fix and issue with resizing the window which would result in cropping the animation. This was solved by specifying a minimum window size so that the user is unable to shrink the window below this size.

09/02/20

Created and input box anchored to the bottom of the window which would later be used to specify the number of passengers used to run the sequence. This could later be expanded to include a count-down for the remaining number of passengers left to transport or implementation of a control panel including more input boxes to edit the number of floors in the building, number of lifts, as well as incorporate buttons for starting, pausing, and autocompleting the sequence. The next thing to do was to ensure that the status bar remained fixed to the bottom of the canvas. Initially I found that whenever shrinking the window the control panel would disappear behind the lift animation.

12/02/20

Started working on the logic behind the user interface. Shifted the random generator to a separate module. This new system would rely on three separate dictionaries where the key to each was the unique id of the number of the passenger. The first dictionary ‘waiting\_dict’ would use this number to access a value containing the users button choice to move either up or down within the building.

Once the lift reaches a passenger the key-value pair is moved from this first dictionary to the ‘moving\_dict’ which signifies that a passenger is inside the lift. From here the value containing up or down is appended with the floor that the passenger needs to be transported to. We can use the length of this dictionary to check whether the lift is over capacity and I will later add an exception method to stop the lift from moving when overweight. The final dictionary ‘arrived\_dict’ will store all passenger that have been transported to their desired floor along with where the passenger came from and is going to, we can once again append the value store by each key with the number of floors that the lift had to travel to serve this passenger. This can be used in later analysis to test the efficiency of the system. From here we can find ratios and averages using the total number of floors travelled by the lift compared to the combined number of floors that all passengers have moved.

14/02/20

There are two different way in which I could link the modules I have designed. One is to allow events from the user interface to effect the logic of the program of alternatively to do the reverse by which every event is triggered directly through the logic module and the user interface in acts passively and only responds to instructions given by this module with no feedback on event states.

The main instance of this that I need to consider is when moving the lift. I can either use for floors in range starting position to end position of passenger travel move the lift in a certain direction. The alternative is to have the user interface handle moving the lift and return to the logic module when done.

15/02/20

I devised an algorithm to get the lift moving. The lift starting from the ground floor check to see if anyone is waiting on this floor and if not moves up one floor and will continue following this process until it comes across a passenger. Once a passenger is collected the lift will continue to their destination floor picking up any additional passenger travelling in the same direction. If a new passengers floor occurs before the old passenger, then the lift will drop them off first and again pick up any passenger travelling in the same direction before carrying the original passenger to their destination. The process repeats over and over until all passengers are server. If the lift has reached the top of the building without picking up and passengers, then it will switch directions. This algorithm is similar to the base case but has the added benefit of being able to change directions whenever a passenger destination is determined. Later on I will change this method to be more active in determining where the lift should travel to next, for example if two people waiting on adjacent floors are travelling in the same direction the lift will later be smart enough to collect both passenger before the first is allowed to change the direction in which the lift travels. Because passengers are not arriving in real time but are instead already generated before the lift begins to move it may later be possible to perform analysis to calculate on average the most efficient route based on probabilities from multiple simulations using only the direction of travel of each passenger. We will have to see how my algorithm develops but even this standard solution is better than the base case. I may need to change my data structure to ensure that the direction of travel can be accessed quickly with the minimal number of operations. Dictionary values which are themselves a list of data may not be the most suitable for this application but ill update this log if I do decide to take a different approach.

28/02/20 – 31/03/20

Between the dates above I spent an average of two hours more or less every day rewriting my program in an Object-Oriented fashion. This increased to closer to five hours almost every day since leaving university for the year on the 17/03/20 as a result of the ongoing situation. I will link below my OneDrive with full version history of my program such that I am able to discuss each in more detail but will provide a brief overview of everything I have achieved in the past month or so.

Starting with versions 30 through 39 I did a complete rewrite of the user interface focussing on validating the users input and allowing the number of people in the building and number of floors to be changed readily using the interface instead of relying on edits to the code. The new interface consists of three columns, one for waiting passengers, one for those travelling in the lift and for passengers delivered to their desired floor. The lift operates as a series of coloured tiles filling the central column where each row splits the column into equally sized cells such that the lift colour can be changed to illustrate that the lift has arrived and departed from a floor. The leftmost column contains the floors numbers and will eventually also include as the number of people waiting on each floor. The final column will later show the number of people delivered to a particular floor.

After having created a template for the interface with most of the elements required for the final version, I started work on the logic of the lift. This is shown in the changes made between versions 40 and 49 of my system.

01/04/20 – 25/04/20

Summary of the past months work:

Versions 50 through 59 details the removal of Floor objects as a means to store waiting passengers. This was changed in favour our two simplified dictionaries, one for waiting passengers and the other for those having been delivered. Where each dictionary key value pair where the key number relates to the floor and the value store is a list containing all the people object waiting to be collected. This was far simpler in the long run and less computationally expensive then having to assign people to a list and keep track of where each list was stored and how to retrieve peoples details such as their originating and destination floor.

Versions 60 through 63 are the development stages require to include a configuration console to allow for the users input of number of passengers, floors, repeats, animation delay, and system logic. This also includes the scripting require to validate each of the users input to ensure that the main logic is able to function correctly and return and error messages to the user before starting the simulation.

Version 63 is the completed version of the basic system.

Versions 70 through 75 are the improved logic where the lift is not impeded by having to reach either end of the shaft before being allowed to change directions. These versions implement two sets, as a means to determine the next useful floor to which the lift should travel, by taking the intersection of these two sets, where if none exists the lift immediately changes directions. This because it is thereby determining that no further operations can be carries out if it were to continue.

Versions 80 through 85 are an attempt at implement a priority queuing system as a mean to further increase the efficiency of the improved system.

This leaves only version 90 through 96 as well as the completed graphical version lift.py. These versions are concerned with logical performance only such that the generation of graphs needed for the statistical analysis can occur as swiftly as possible. To this end version 96 is a version of the code incorporating both the basic and improved version of the building’s main logic such that the two can be compare side by side and plotted on the same axis.

The final version of the code lift.py is simple a combination of the two systems in one file such that it can be easily appended to the report. For this version I have also finally got around to making a functioning toggle switch to choose between both of the different lift system such that the user can select which they would prefer to see animated.