Austin G. Walters

Everyday Algorithms: Elevator Allocation

July 14, 2014 Austin Comments are off for this post.

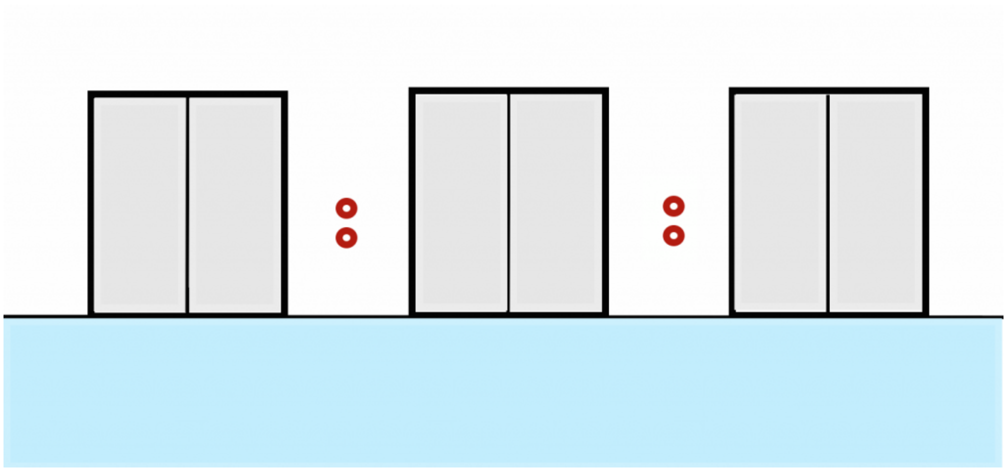
Every day in cities such as Chicago, New York, Tokyo, Singapore, Hong Kong and more, millions of people attempt to leave their buildings via elevator. However, very rarely do we consider how elevators are allocated to provide service, especially during rush hour(s), when most of the building will attempt to exit in a matter of an hour or so.

There is at least one patent on an algorithm on the subject (Elevator call allocation system based upon passenger waiting time), research (Elevator Traffic Simulation), and has appeared on Quora as well. I have even been asked how I would allocate elevators on an interview before (which I wrote about previously):

> *There are ten floors, each with the same number of people living on each floor. There are three elevators and no stairs. How would you allocate the elevators for optimal(ish) performance, minimizing waiting time for each floor?*

I enjoyed this interview question, I thought it was both challenging and you could take it as far as you desired, but also straight forward enough that you could "get started" and produce some sort of solution. Rather than trying to rack my brain for a month trying to model a real world scenario, in this article I will attempt to solve a *simplified* model of allocating elevators, similar to the interview question above.

Problem Description



Creating an algorithm applicable for the real world allocation of elevators is fairly difficult (and is apparently patented). Therefore, I will be trying to solve something similar to my interview question, with a few minor changes:

> *Design an algorithm to minimize the total waiting time of all individuals waiting in a building, while also taking into consideration load per elevator. Given there are equal number of people on each floor, with a uniform appearance of individuals to use the elevator at each floor. Assuming there are several hours a day which are "rush hour times," the algorithm should provide the most "fair" way to distribute the elevators to the various floors.*

That was a mouthful, but if we break it down, the problem consists of the following:

- Arbitrary number of floors
- Arbitrary number of elevators
- Given rush hour times
- We must distribute elevators based on some function of load and time

There are also some unsaid variables/constants we should take into consideration:

- Number of individuals per floor: 100 persons
- Time for the elevator to transition one floor (without stopping): 5 seconds
- Waiting time per floor: 20 seconds

I added numbers to the variables above, and although "time for the elevator to transition one floor" is likely not linear (that is to say, it takes time to accelerate from a stopped position), we will assume so in this case. Making these assumptions may "over simplify" the problem, however I believe this article will suffice for an interview or as an excellent launching point to a more in-depth contemplation/discussion.

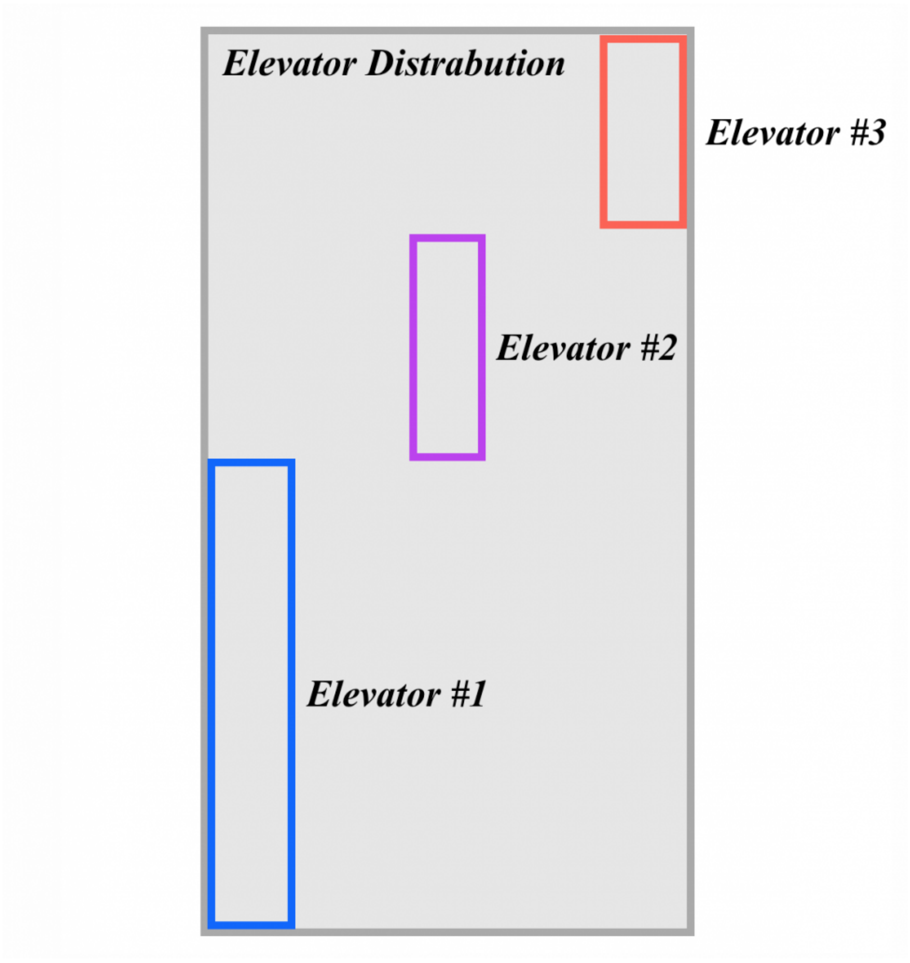
Notice that I did not take into consideration the capacity of the elevators, in this case I am going to make a *big* assumption. My assumption will be (throughout this solution) that each elevator has a limitless capacity. Obviously, this cannot be true, however once we have a solution in hand I feel it will be simpler to add a statement such as:

> If elevator is full, return to lower level
> After dropping off guests, return to previous floor

Perhaps, I will write another article and post it on this blog or I will distribute via my mailing list, in either case, I hope someone will try to solve it themselves!

Elevator Allocation Algorithm

> *This is probably not the best solution,*
> *though it does seem to work out well.*
>
> *If you find a better solution, please share!*



As the image above suggests, I decided to designate a specific elevator to a particular floor, I'll call this *zone elevator allocation*. The idea being, we can attempt to average the wait time of each floor as well as the load of each elevator.

My particular solution was based off a few observations I made on the time it takes for each elevator to make a circuit (go through all the floors in its loop, i.e. 0 -> 1 -> 2 -> 0). All we need to know is the following to calculate the time for an elevator to complete a circuit:

1. Time to pass a floor times maximum floor in the circuit times two (up and down), in this case: (5 seconds * <maxFloor> * 2)
2. Number of stops between ground level (0) to the maximum floor in the circuit times the waiting time at each stop, in this case: (20 seconds * <floorsServiced>)

That's all it takes to calculate time, all together:

elevatorsCircuitTime=(5*<maxFloor>*2)+(20*<floorsServiced>)

To then calculate the *average* number of people we carry per circuit we use the following equation:

> avgElevatorLoad=<elevatorsCircuitTime>*<floorsServiced>*<peoplePerFloor>/<rushHour>

The variables *rushHour* equate to the time it takes to complete one rush hours, *floorsServiced* are the number of floors the elevator stops at, *peoplePerFloor* are the number of individuals on a given floor, and we already calculate *elevatorsCircuitTime*. We can then use the time and the average load capacity to complete our algorithm.

My solution to this problem requires two arrays:

1. Representing the building: The array is filled with the number of people per floor. Each slot in the array representing a floor. An example would be [100, 100, 100] would be a four story building with only the top three floors being people who need elevators.
2. Representing the elevators: Each slot in the array represents the "maximum" floor which that elevator will reach on its circuit (I also input '0' in the first slot in the array for simplicity). For example, [0, 2, 3] represents an two elevators, elevator one (in slot one) carries passengers from floor 2 and 1 to 0. Elevator two (in slot two) carries passengers from only floor 3 to 0.

My solution begins with array 1 (representing the building) being empty, then every time I "add a floor" to the array I assign an elevator to the floor. The assignments can change, as you will see, but it does follow a similar pattern. The elevator circuit with the lowest circuit is assigned the new floor, unless the capacity is becoming an issue. I added a little function:

> elevatorCircuitTime + ((elevatorCircuitTime / 100) * elevatorsAvgLoad)

Because *elevatorCircuitTime* is an integer, unless the circuit time exceeds 100 seconds (which is a long time for being in an elevator), then the *elevatorsAvgLoad* beings being factored into the equation. The problem description was rather vague, and as such my solution does have some vagueness regarding the equation above.

As such, the function I used to assign a floor to an elevator is rather arbitrary, but it was effective at load managing (though there is likely a *more* optimal function).

The implementation of the function to add a floor is below:

```python
# (Index * 5 seconds) + (20 seconds * (Index - PrevIndex))
# If previous elevators loops/stops add up to be greater than,
# (timePerFloor * 2) + timePerWait, then increase floor of previous
# elevators loop. i.e. elevator[2]+=1
# e represents elevatorArray
def addFloor(e):
    best = 99999
    for i in range(1, len(e)):
        cirTime, avgCarry = eleLoop(e, i)
        if cirTime + ((cirTime / 100) * avgCarry) < best:
            elevatorNumber = i
            best = cirTime + ((cirTime / 100) * avgCarry)
    for i in range(elevatorNumber, len(e)):
        e[i] += 1
    return e
```

addFloor.py hosted with ❤ by GitHub view raw

Notice, every time a "elevatorNumber," is chosen every elevator above the "elevatorNumber" the elevator array is then bumped up one:

> for i in range(elevatorNumber, len(e)):
>
> e[i] += 1

This is because the *maximum floor* in each circuit above the chosen elevator is still being increased by one, we only want to add one additional elevator to the chosen elevators circuit. The additional function *eleLoop(e, i)* simply determines the time and average number of people being carried on the circuit.

Once we have the function to add floors, I created a function to loop through and create the floors. Note, in this case all the floors are assumed to be uniform, if this problem was to be expanded upon variable number of people per floor would be fairly easy to take into account.

```python
# Allocate elevators
# Elevator[] represents the starting
# group of stops.
def elevatorAllocation(building, elevatorCount):
    elevator = []
    for i in range(elevatorCount + 1):
        elevator.append(0)
    for i in range(1, floorCount):
        elevator = addFloor(elevator)
    printeleLoop(elevator)
```

elevatorAllocation.py hosted with ❤ by GitHub view raw

That's pretty much it for the allocation portion of the algorithm. It is relatively straight forward and has a fair amount of room for possible improvements, which I leave to you!

Implementation | Python Code

If we piece all of the various portions of the algorithm together, sprinkle in a few extra functions to print out the data and build a little simulator we get a pretty cool little program (which you can fork/view on my github).

Variables:

- 10 Floors
- 3 Elevators
- 1 Rush Hour
- 5 seconds to pass a floor
- 20 seconds to stop at a floor
- 100 people per floor

```python
# Sets up the building, filling all the floors with people
def fillBuilding():
    building = []
    for i in range(floorCount - 1):
        building.append(peoplePerFloor)
    return building

# Determines the time for circuit (cirTime),
# as well as average carrying capacity per circuit.
# Given e - array of elevators, which holds the highest
# serviced floor, and i the current index of e.
```

```
def eleLoop(e, i):
    floorsServiced = e[i] - e[i-1] + 1
    cirTime = timePerFloor * e[i] * 2
    cirTime += timePerWait * floorsServiced
    avgCarry = cirTime * peoplePerFloor / rushHour * floorsServiced
    return cirTime, avgCarry

# (Index * 5 seconds) + (20 seconds * (Index - PrevIndex))
# If previous elevators loops/stops add up to be greater than,
# (timePerFloor * 2) + timePerWait, then increase floor of previous
# elevators loop. i.e. elevator[2]+=1
def addFloor(e):
    best = 9999
    for i in range(1, len(e)):
        cirTime, avgCarry = eleLoop(e, i)
        if cirTime + ((cirTime / 100) * avgCarry) < best:
            elevatorNumber = i
            best = cirTime + ((cirTime / 100) * avgCarry)
    for i in range(elevatorNumber, len(e)):
        e[i] += 1
    return e

# Prints the population of the buildings floor as an array.
def printApprox(building):
  str = '[  '
  for i in range(len(building)):
      str += '%06.3f  ' % building[i]
  str += ']'
  print str


# Prints the circuit(s) for each of the elevators
def printeleLoop(e):
    print ''
    print e
    print ''
    for i in range(1, len(e)):
        floorsServiced = e[i] - e[i-1] + 1
        curr = timePerFloor * e[i] * 2
        curr += timePerWait * floorsServiced
        avgCarry = curr * peoplePerFloor / rushHour * floorsServiced
        str = 'Elevator #%d, time for loop %d seconds, ' % (i, curr)
        str += 'carrying an average of '
        str += '%3.2f people per carry' % avgCarry
        print str
    print ''

```

```
# Allocate elevators
# Elevator[] represents the starting
# group of stops.
def elevatorAllocation(building, elevatorCount):

    elevator = []
    for i in range(elevatorCount + 1):
        elevator.append(0)
    for i in range(1, floorCount):
        elevator = addFloor(elevator)
    printeleLoop(elevator)
    return elevator

# Simulates the building being emptied at rush hour
def simulate(e, building):

    str = '[  '
    for floor in range(len(building)):
      str += 'floor%2d ' % (floor + 1)
    str += ']'
    print str

    eCircuit = []
    for i in range(len(e)):
        curr, avgCarry = eleLoop(e, i)
        eCircuit.append(float(curr))

    emptyFloors = 0
    iteration = 0
    finalFloor = 0

    while emptyFloors < len(building):
        emptyFloors = 0
        iteration += 1
        for i in range(1, len(e)):
            for j in range(e[i-1], e[i]):
                if building[j] > 0.0:
                    persons = eCircuit[i] * peoplePerFloor / rushHour
                    building[j] = building[j] - persons
                if 0 >= building[j]:
                    building[j] = 0.0
                    emptyFloors += 1
                    finalFloor = j
        printApprox(building)
    print ''

    # Find the final elevator on circuit, prints time
    for i in range(len(e)):
        if e[i] > finalFloor:
            iteration = eCircuit[i] * iteration / 60
    print 'Total Time: %d minutes\n' % (iteration)

# ___ MAIN ____
building = fillBuilding()
elevator = elevatorAllocation(building, elevatorCount)
simulate(elevator, building)
```

elevatorAllocation.py hosted with ❤ by GitHub view raw

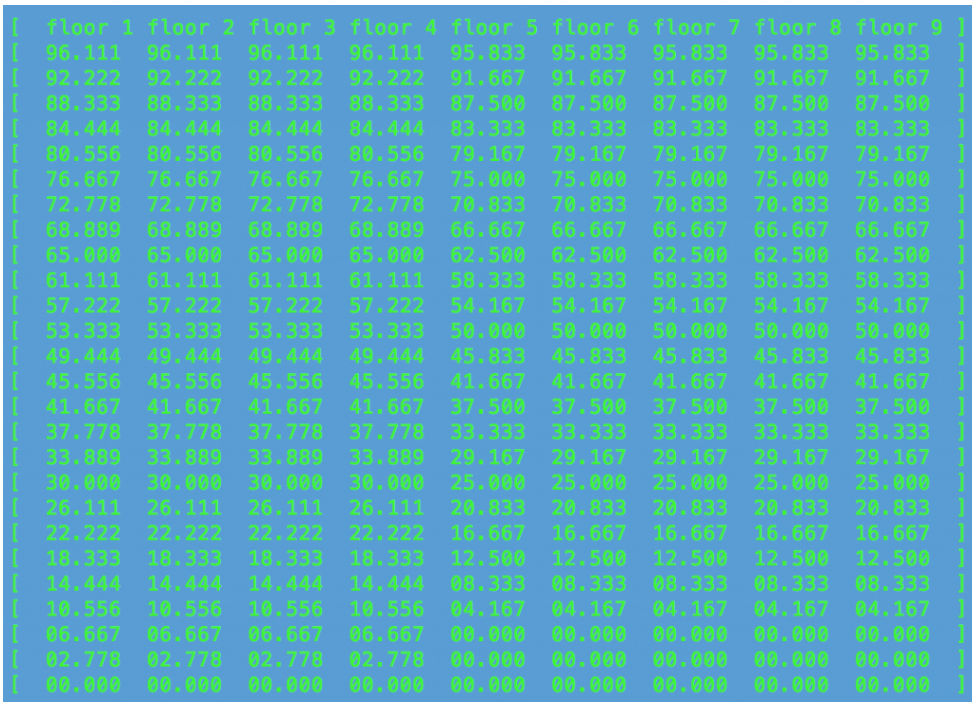
Output:

```
[0, 4, 7, 9]

Elevator #1, time for loop 140 seconds, carrying an average of 19.44 people per carry
Elevator #2, time for loop 150 seconds, carrying an average of 16.67 people per carry
Elevator #3, time for loop 150 seconds, carrying an average of 12.50 people per carry

Total Time: 65 minutes
```

Average number of people per floor as the elevators iterate:



```
[  floor 1 floor 2 floor 3 floor 4 floor 5 floor 6 floor 7 floor 8 floor 9 ]
[  96.111  96.111  96.111  96.111  95.833  95.833  95.833  95.833  95.833  ]
[  92.222  92.222  92.222  92.222  91.667  91.667  91.667  91.667  91.667  ]
[  88.333  88.333  88.333  88.333  87.500  87.500  87.500  87.500  87.500  ]
[  84.444  84.444  84.444  84.444  83.333  83.333  83.333  83.333  83.333  ]
[  80.556  80.556  80.556  80.556  79.167  79.167  79.167  79.167  79.167  ]
[  76.667  76.667  76.667  76.667  75.000  75.000  75.000  75.000  75.000  ]
[  72.778  72.778  72.778  72.778  70.833  70.833  70.833  70.833  70.833  ]
[  68.889  68.889  68.889  68.889  66.667  66.667  66.667  66.667  66.667  ]
[  65.000  65.000  65.000  65.000  62.500  62.500  62.500  62.500  62.500  ]
[  61.111  61.111  61.111  61.111  58.333  58.333  58.333  58.333  58.333  ]
[  57.222  57.222  57.222  57.222  54.167  54.167  54.167  54.167  54.167  ]
[  53.333  53.333  53.333  53.333  50.000  50.000  50.000  50.000  50.000  ]
[  49.444  49.444  49.444  49.444  45.833  45.833  45.833  45.833  45.833  ]
[  45.556  45.556  45.556  45.556  41.667  41.667  41.667  41.667  41.667  ]
[  41.667  41.667  41.667  41.667  37.500  37.500  37.500  37.500  37.500  ]
[  37.778  37.778  37.778  37.778  33.333  33.333  33.333  33.333  33.333  ]
[  33.889  33.889  33.889  33.889  29.167  29.167  29.167  29.167  29.167  ]
[  30.000  30.000  30.000  30.000  25.000  25.000  25.000  25.000  25.000  ]
[  26.111  26.111  26.111  26.111  20.833  20.833  20.833  20.833  20.833  ]
[  22.222  22.222  22.222  22.222  16.667  16.667  16.667  16.667  16.667  ]
[  18.333  18.333  18.333  18.333  12.500  12.500  12.500  12.500  12.500  ]
[  14.444  14.444  14.444  14.444  08.333  08.333  08.333  08.333  08.333  ]
[  10.556  10.556  10.556  10.556  04.167  04.167  04.167  04.167  04.167  ]
[  06.667  06.667  06.667  06.667  00.000  00.000  00.000  00.000  00.000  ]
[  02.778  02.778  02.778  02.778  00.000  00.000  00.000  00.000  00.000  ]
[  00.000  00.000  00.000  00.000  00.000  00.000  00.000  00.000  00.000  ]
```

As you can see my algorithm provides a *good*, if not the best solution (in this case) and although there is always room to improve (and I challenge you to do so) it is a good start.

Calculating Runtime

The runtime and space required for this algorithm are a little difficult to calculate, but not overly so. The runtime of this algorithm is based off three factors:

> k: longest circuits, number of people
> n: starting number of people in on the longest circuits serviced floor (that's a tung twister)
> m: total number of floors

- Run Time: O(m * (n/k))

'n / k', because that determines the maximum number of iterations the elevators make on the circuit(s) and 'm' because we have to iterate over each floor ever iteration of the elevators. In this case we neglect the "setup" which is filling the "building" array, representing the number of people on each floor, since it is not the dominant term for the run time (m*(n/k) + m).

The maximum space required is *very* straight forward:

> e: number of elevators
> f: number of floors

- Memory Required: O(e + f)

If we brought it all together:

> k: longest circuits, number of people
> n: starting number of people in on the longest circuits serviced floor
> m: total number of floors
> e: number of elevators
> f: number of floors

- Run Time: O(m * (n / k))

- Memory Required O(e + f)

Closing Remarks

I recognize this is not the *optimal* solution, however it does get the job done. I challenge you to comment, fork from github and improve my code, or write your own solution in your article. I think it is an interesting problem and is similar to resource allocation inside your computer, I wrote a basic introduction to process scheduling as well, if you are interested. I would love to see a different (hopefully better) solution, so if you come up with one don't forget to write!

I do plan on writing another article diving more in-depth into this problem and proving a better *real world* applicable algorithm, however I do not have a date set yet (could be days, weeks, months, or years). I hope you enjoyed the article and would love to hear your thoughts, so don't hesitate to comment or email me, thank you!

Related Articles

- I/O Multiplexing using epoll and kqueue System Calls
- Everyday Algorithms: Pancake Sort
- Counting Sort in C
- Radix Sort in Go (and C)
- Analytics, Experimentation, and Results of Building a Blog: Month Three

Recommended Reading

- Introduction to Algorithms (Cormen, Leiserson, Rivest, Stein)
- Algorithms (Jeff Erickson)
- Introduction to Theory of Computation (Sipser)

Categorized in: Today I Learned