Safe and Cost-Effective Strategies for Stunt Driving: Determining the Highest 'Safe' Floor for Car Drops

Jet Hughes, Callum Teape, Cody Airey, Jake Norton

June 4, 2023

Contents

1 Introduction

This investigation concerns the world of Hollywood stunts, where size and spectacle are given top priority. However, amplified scale always results in increased costs and potential risks. In this case, we will be investigating the feasibility of driving a car off a 100-storey building—with the car remaining intact.

The primary aim of this investigation is to create a method to determine the highest 'safe' floor from which the stunt can be done, while ensuring the car remains intact. We will consider the cost of conducting the tests and the number of cars required. By gaining an understanding of the inherent risks and costs of this stunt, we hope to provide valuable insights to help with the decision-making for stunt planning and execution.

2 Question 1

2.1 Initial approach: Binary search

To determine the highest "safe" floor, we can use a binary search algorithm. We start by dropping a car from the middle floor (50th floor) of the building. If the car survives the drop and can be driven away, we know that all the floors below the 50th floor are safe. In this case, we continue the search on the upper half of the building (51st to 100th floor). If the car doesn't survive, we know that all the floors above the 50th floor are unsafe, and we continue the search on the lower half (1st to 49th floor). We repeat this process, dividing the remaining floors in half each time, until we narrow down the range to just one floor. This will give us the highest "safe" floor. The worst-case scenario would be if the highest "safe" floor is the 100th floor, which would require performing the binary search $\log2(100) = 7$ times.

However, this method doesn't work. For example: Say the highest safe floor is 12. We drop the first car at 50 and it breaks. Then we drop the second car at 25 and it also breaks. Now we are left with two wreckages and still don't know the highest safe floor. To fix this issue, we devised a second solution.

2.2 Revised Solution: Incremental Testing

This method works by dropping the cars from incrementally higher floors until it breaks. When the first car breaks, we have a range in which the highest floor must lie. We then use the second car to test each floor in the range, starting at the lowest floor.

With this method, we always break both cars unless the highest floor is floor 100 in which case we break zero cars, or the highest floor is floor 99 in which case we only break one car.

This method results in a reasonably low average case. The worst case for this method is when the highest floor is floor 98 where we drop the first car 9 times to determine the range 100-90. Then drop the second car 9 further times, which breaks on floor 99, to find the highest 'safe' floor is 98.

The second car will always be a linear search incrementing by 1. In order to optimise this solution, we need to determine the optimal sequence of floors to drop the first car. This could either increment linearly from a starting point, or it could be something else. To find the optimal value for a linear incremental approach.

We created a simple python script to simulate this method with a given increment size.

```
def incremental_search(highest_safe_floor, floor_increment):
    total_cost, current_highest = 0, 0
    cars_remaining = 2
    while(True):
        total_cost += DROP_COST # We always pay the drop cost
        # Check if the car will break when we increase the floor
        if(current_highest + floor_increment > highest_safe_floor):
            cars_remaining -= 1 # We broke a car
            total_cost += DROP_COST
            if(cars_remaining == 0): # If we have no more cars
                return total_cost # We found the highest floor
            floor_increment = 1 # Start moving one floor at a time
        else: # If the car wont break move up to the next floor
            current_highest += floor_increment
```

We ran this script with increment sizes of 1, 2, 4, 10, 20, 25 and 50. The results of this are shown in Figure 1. We can see that very extreme increments are not good and that the best worst-case scenario occurs with an increment of 10. In Figure 2 we can see a line plot of the cost of the worst-case scenario for every increment from 1-100. This graph shows a clear minimum at 10 with a sharp decrease in cost from 0-10 and a near linear increase from 10 upwards.

Although this approach seemed quite good, we thought that it could still be improved. If the car breaks at floor 10 we only need to drop the car 9 more times if the highest floor is 10. Similarly, if the car breaks at floor 100, we need to drop the car 9 more times if the highest floor is 100. But in the second case, we have already dropped the car 10 times, giving us a total of 19 drops. In order to improve the solution, we could somehow change our approach so that regardless of where the first car breaks, the total number of drops in the worst case is the same. We need to somehow spread the extra nine drops from the worst case of 19 across the other drops.

2.3 Final Solution: Altered Incremental Testing

We believe this is the optimal solution. Instead of incrementing by the same amount at each step, by starting at a higher initial floor, and decreasing the amount we increment each time, we are able to decrease the cost of the worst-case scenario.

The worst case of the previous approach was dropping the car 19 times. To improve this, if we drop the first car at the 18th floor, at worst we have to drop it 17 more times (at 1, 2, 3 ... 16) to find out that the highest floor is the 18th floor, giving us 18 drops. Then instead of doing the second drop at floor 36 (18+18) we do it at floor 35 (18+17), and in the worst case we have to drop it 16 more times, which gives us the same worst case of 18 drops.

If we repeat this process up to the 100th floor, the worst case is always 18. Now all there is to find what floor is best to drop the first car. To test this, we wrote a simple python program[1]. It is very similar to the previous method, except we increment by one less each time.

[1]This is a simplified version of the method. The full method has some checks to avoid edge cases when we reach the top of the building. It can be found in linearSearch.py at https://altitude.otago.ac.nz/jetcosc326/11-dropping-cars

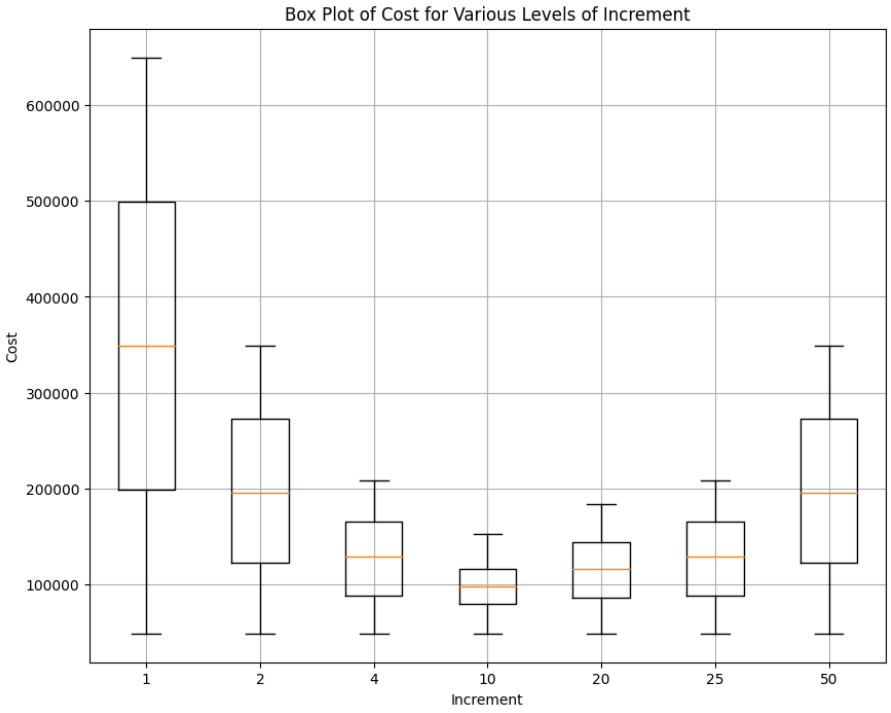


Figure 1: Cost vs Highest safe floor using Linear Search with various Increments

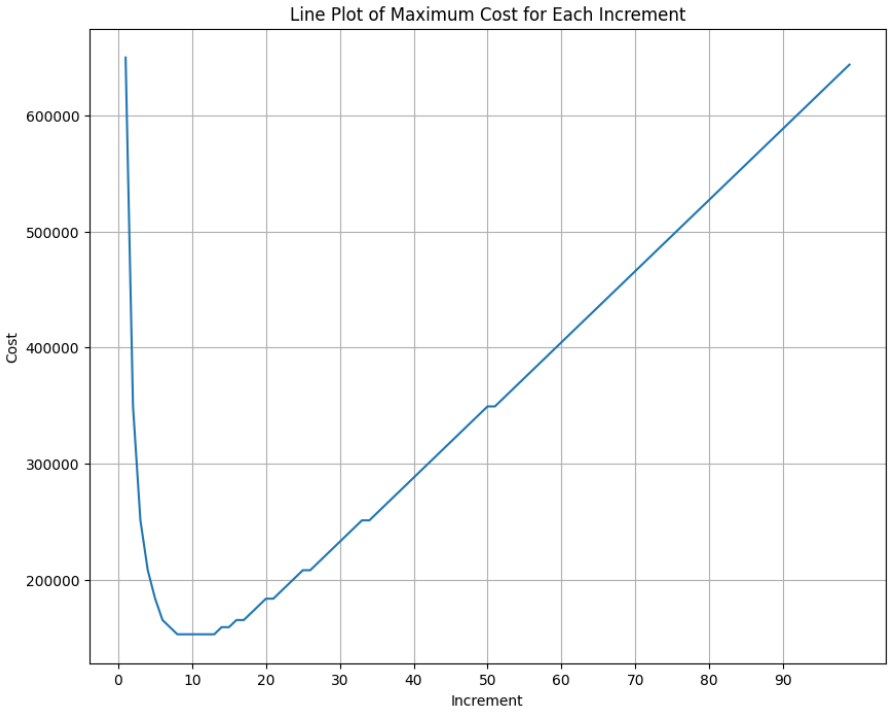


Figure 2: Line Plot of Maximum Cost for Each Increment

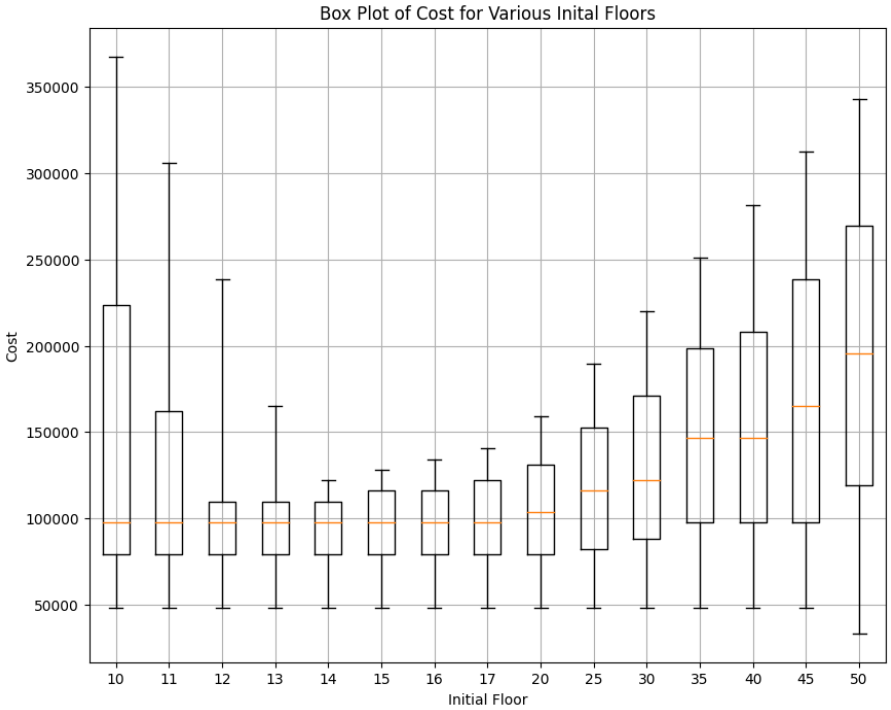


Figure 3: Box Plot of Cost for Various Initial Floors

```
def adjusting_incremental_search(highest_safe_floor, initial_floor):
    total_cost, current_highest, = 0, 0
    increment = initial_floor
    cars_remaining = NUMBER_OF_CARS
    while(True):
        total_cost += DROP_COST # We always pay the drop cost
        # Check if the car will break when in increase the floor
        if(current_highest + increment > highest_safe_floor):
            cars_remaining -= 1 # We Broke a car
            total_cost += CAR_COST
            if(cars_remaining == 0): # If we have no more cars
                return total_cost # We found the highest floor
            increment = 1 # Start checking one floor at a time
        else: # If the car doesn't break move up to the next floor
            current_highest += increment
            increment -= 1 # For each drop, the increment gets smaller
            if increment < 1 : increment = 1 # The increment should never be less than 1
```

We ran this function for every possible target floor using a variety of Initial Floors, producing the graph in Figure 3. From this, we can see that the lowest worst-case occurs when we use an Initial Floor of 14. Figure 4 Show the cost at each floor for this method starting at floor 14 versus the cost at each floor of the original method incrementing by 10. This graph clearly shows the new method is limits the worst case.

This method gives us a worst-case of $122068

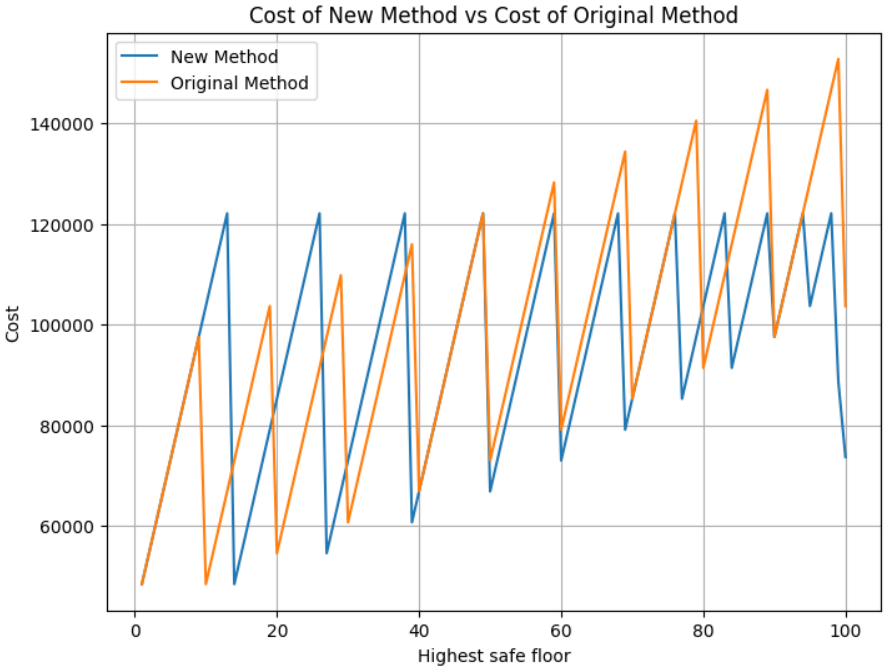


Figure 4: Cost of New Method vs cost of Original Method

3 Question 2

If we are allowed more cars, the method will certainly change. In the process of find a method for 100 stories where we are allowed more cars, we came to the general solution with a worst-case cost of $103654. This is much better than the method with only two cars.

4 Question 3: General Method

4.1 First Approach: Adjusted Binary search

We started by considering the behaviour at the two extremes: Firstly, what would the best strategy be if there was no additional cost for breaking a car? And secondly, what would the best strategy be if the cost of breaking a car approaches infinity?

For the first question, the answer would be a binary search tree, where we start by dropping at floor 50, if the car breaks then move to 25, if it doesn't then move to 75. If the car breaks at 25, move to 12, if not move to 38, and so on. For consistency, we decided to always round the left node down to the nearest integer, and round the right node up. Using this algorithm with 100 floors, on average it should take roughly $log_2(100) = 6.644$ drops to determine the highest safest floor. The worst case must be if we break the car on every drop, which would be the path in the tree consisting entirely of left turns: 50, 25, 12, 6, 3, 2, 1. If we break on every one of these, then the highest safest floor is 0, and the cost of the algorithm is $7(14999 + 6138) = 147959$.

To answer the second question, as the cost of breaking a car tends to infinity, it is always more cost-effective to only break the minimum possible number of cars, regardless of how many drops you have to do. For instance, it is cheaper to successfully drop 100 times, then break a car once. Therefore, the answer would be a simple linear search, where we start at the first floor and increment by one until we break a car. The worst case for this algorithm would be if the highest safe floor was the 99th floor. Then we would have to drop at all 100 floors, breaking on the last one.

This gave us the idea that the binary and linear search can be placed on a spectrum, whereas the cost of breaking a car increases it becomes more efficient to use a more linear search. The question now was, how can we modify the binary search tree to become more like linear search?

To start, we looked at the tree diagram for a regular binary search tree.

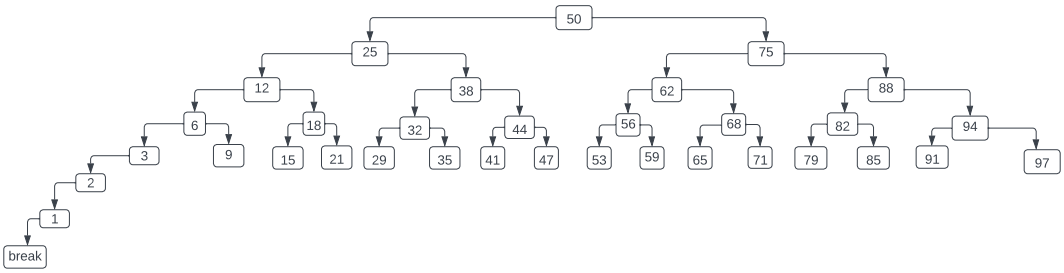


Figure 5: Binary Search Tree Diagram

Again, the most expensive path is the path consisting of entirely left turns, which we calculated as costing \$147959. We started by comparing the cost of a left turn with the cost of a right turn. A left turn is $\frac{14999+6138}{6138} = 3.44$ times more expensive than a right turn. Using this, we decided that if we can replace one of the left turns in the most expensive path with three or less right turns, then we have reduced the worst case cost. So we decided to start the search at floor $\frac{100}{3} = 34$ (rounding up) instead, and conduct a sort of ternary search from there. The tree diagram (with less important branches omitted) for this algorithm is given below:

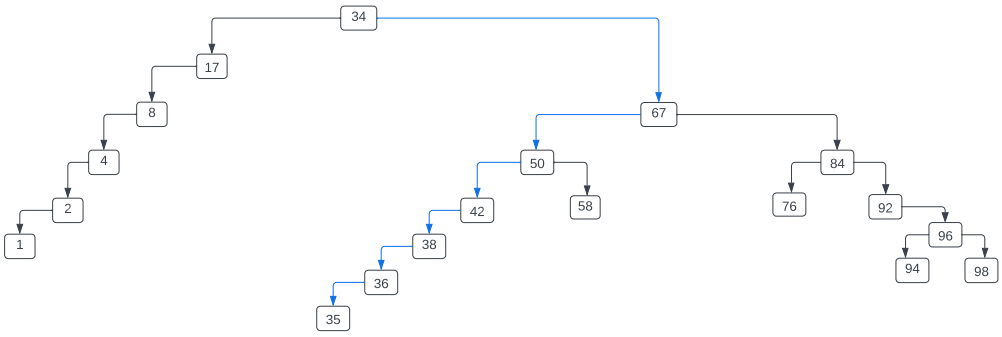


Figure 6: Ternary Search Tree Diagram

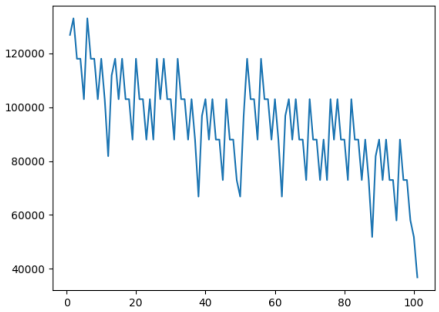
As we can see, the most expensive path (highlighted in blue), now consists of 1 right turn and 6 left turns. So we succeeded in replacing one of the left turns with a right. This makes the most expensive path 14999 cheaper, meaning the new worst case cost is now \$132960.

So we kept repeating this process, constructing a tree starting at $\frac{100}{4} = 25$, then at $\frac{100}{5} = 20$ until we got to the point where we had to start adding more than 3 right turns to replace a left turn. We can think of this process as rebalancing the tree until the cost of each path is as similar as possible. To save time, we decided to do this using the following code:

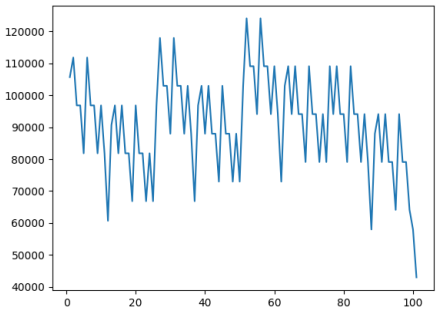
```
def constructTree(firstDrop, maxSafe, n, dropCost, carCost):
```

```
    """
    :params firstDrop: The first floor we drop from
    :params maxSafe: the maximum floor a car can be dropped from without breaking
    :params n: the number of floors in a building
    :params dropCost: the cost of successfully dropping a car (without breaking)
    :params carCost: the cost of breaking a car
    """

    firstLeft = False
    tested = []
    current = math.ceil(gapSize)
    prior = 0
    width = current
    leftTurn = 0
    rightTurn = 0


    while not (width <= 3):
        tested.append(current)
        prior = current
        if current <= maxSafe:
            if (not firstLeft and n-current > gapSize):
                current = min(100, math.ceil(current + gapSize))
            else:
                current = min(100, math.ceil(current + width/2.0)) #NOTE THIS IS THE WIDTH
                    FROM THE PREVIOUS ITERATION
            rightTurn += 1

        else:
            current = max(1, math.floor(current - width/2.0)) #NOTE THIS IS THE WIDTH FROM
                THE PREVIOUS ITERATION
            firstLeft = True
            leftTurn += 1

        width = abs((prior - current)) #updating width


    while(not current in tested and not current < 1 and not current > 100):
        tested.append(current)
        if current <= maxSafe:
            current += 1
            rightTurn += 1
        else:
            current -= 1
            leftTurn += 1

    return leftTurn*(dropCost + carCost) + rightTurn*(dropCost)
```

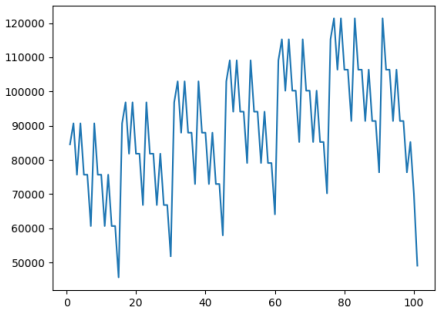
We called this method 101 times, with safest floors ranging form 0 to 100 to figure out the average and worst case cost at each starting point. Below are the graphs of the cost for each maximum safest floor for starting points 50, 25, 16, and 12:



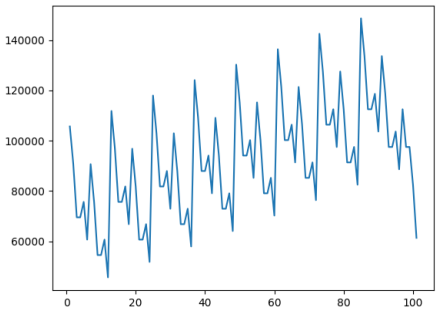
(a) first drop = 50



(b) first drop = 25



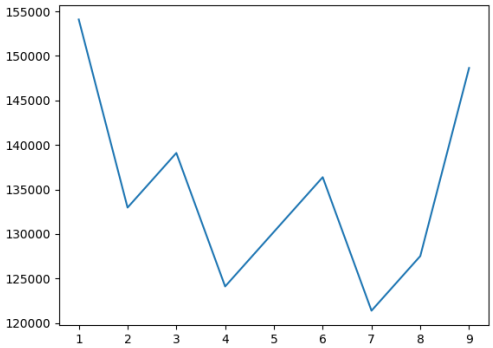
(c) first drop = 16



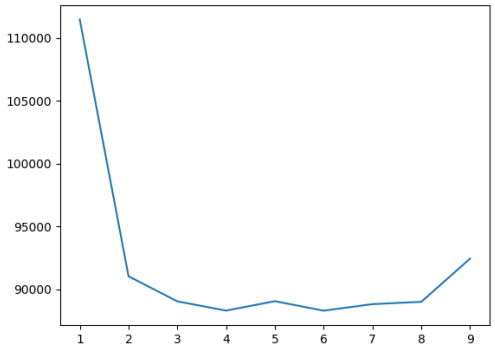
(d) first drop = 12

We notice a few things about these graphs. Firstly, we can see that the graphs become increasingly linear as the first drop gets lower. This reflects our goal of 'linearising' the binary search tree. We can also see the bifurcations in the tree reflected in the spikes in the graphs.

We then obtained the mean and maximum of the costs for each n/k and found the following:



(a) maxes



(b) means

From the graph, we can see that the cheapest average and worst case scenario roughly occurs when the first drop is placed at $\frac{100}{7} = 15$ (rounding up). This gives a worst case of $121376 at floor 76, and an average case of $88828. Linearising the tree further than this gives worst results, so we think that this is the point where we have to start adding more than three right turns to replace a left turn. This is therefore the most cost balanced tree we can build.

4.2 Second approach: Weighted Binary Search

Our second approach was based on probability. The ratio of the cost of dropping a car to the total of the possible costs gives us the probability we want for the car to survive.

$$\frac{dropcost}{(carcost + dropcost) + dropcost}$$

$$\frac{6138}{(14999 + 6138) + 6138} = 0.22(2dp)$$

From this, we know to drop the first car at the 22nd floor. Then, if it breaks, we drop it again at the 0.22 * 22 floor. If it survives, however, we drop the second car at the 22 + (78 * 0.22) floor. Essentially, the value 0.22 defines a weighted midpoint for a binary search.

Again, we wrote a python script to simulate this method. It is a modified recursive binary search algorithm[2].

```
def weightedBinarySearch(target, bottom, mid, top, total_cost):
    if (bottom == top-1): # Base case, we have found the target
        return bottom, total_cost
    total_cost += DROP_COST # Always add the drop cost
    if (mid > target): # Car Breaks - search again in the bottom "half"
        total_cost += CAR_COST
        top = mid
        mid = math.ceil(bottom + ((top - mid) * WEIGHT))
    elif (mid <= target): # Car survives - search again in the top "half"
        bottom = mid
        mid = math.ceil(mid + ((top - mid) * WEIGHT))
    mid = clamp(mid, bottom+1, top) # clamp mid between bottom+1 and top
    return weightedBinarySearch(target, bottom, mid, top, total_cost) # Recursive call
```

We ran this method for every floor in the building, giving the results in Figure 7. This method has a worst-case cost of $103654 and an average-case cost of $84441.16

5 Question 4

The worst case with a 92-storey building is at floor 88. The worst-case cost is $103654, and we only break two cars. The average-case cost of $82748.40 is slightly less than with 100 stories. This is shown in Figure 8

[2]Code is simplified for readability. The full weightedBinarySearch method can be found in weightedBinarySearch.py at https://altitude.otago.ac.nz/jetcosc326/11-dropping-cars

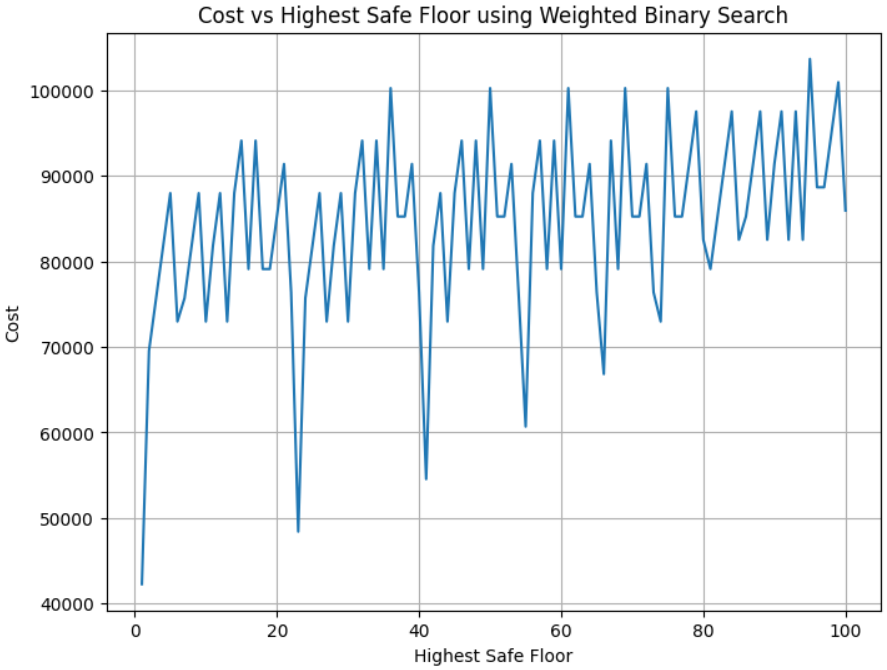


Figure 7: Cost vs Highest Safe Floor using Weighted Binary Search

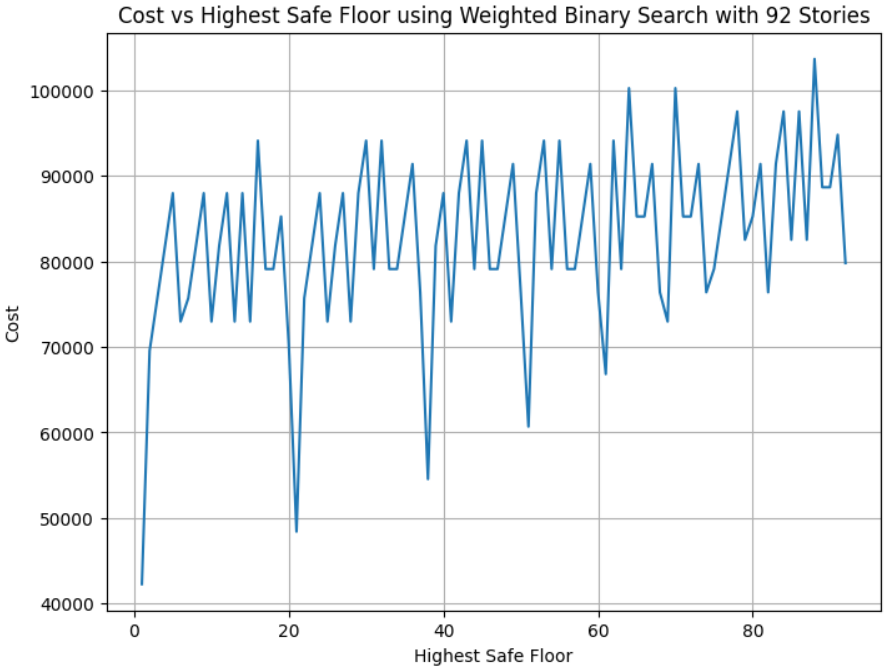


Figure 8: Cost vs Highest Safe Floor using Weighted Binary Search with 92 Stories