The Egg Dropping Problem

Dan Van

9 November 2025

Abstract

The two eggs drop problem is an old problem that was often attributed to Google or Microsoft for coding interviews. Here, we plan to generalize this problem, expand on the premise, and prove that our solution is optimal. These approaches to expand include: arbitrary eggs and floors, considering approximating solutions. We then provide a quantum algorithm that performs better than the classical lower bound to demonstrate quantum advantage.

1 Introduction

The original problem statement goes along these lines. You have 2 eggs, and you're in a building. The goal is to drop these 2 eggs off the balcony of the building to determine which floor is the highest floor on which an egg can be dropped and it would not crack. We shall call this floor the threshold. Adopting some common sense from [4], we can make the following assumptions:

1. If an egg breaks when dropped, then it would break if dropped from a higher floor.
2. If an egg doesn't break when dropped, then it would not break if dropped from any lower floors.
3. An egg that survives a fall can be used again.
4. A broken egg must be discarded.
5. The effect of a fall is the same for all eggs.

We also assume that we can instantly teleport between floors at no cost, and the only cost is dropping the eggs.

2 Related Work

There are previous work on solving this problem exactly. GeeksforGeeks provided an algorithmic approach to calculate the exact worst case. It utilizes dynamic programming to get the lowest query count in the worst case in $O(n \cdot e)$ time, where n is the number of floors, and e is the number of eggs [2]. DataGenetics provided a graphical representation of the worst case [1].

For our contribution, we provide a proof of the asymptotic lower bound, and provide confirmation for the DataGenetics graphical representation that the functions do indeed follow the trend. We also provide a quantum algorithm that demonstrates advantage over any classical algorithm.

3 Background

This problem is equivalent to the following problem. There is a secret, n-bit string s of the form

$$s = 0^m 1^{n-m}$$

And the goal is to find m. The algorithm can query at positions p of s. The algorithm only has d chances to get back $s[p]$ is 1, where $s[p]$ denotes the p-th bit of the string s. In the original problem, d is 2. One may pose the same question with determining m as distinguishing between the strings

$$s_m = 0^m 1^{n-m}$$

and the goal is to output the full string s_m. This shall be useful to determine when an algorithm can decide m with certainty.

It is clear that this problem exactly mimics the egg dropping problem. The string s encodes the floor dropping information as for any i such that $s[i] = 0$, then $s[j] = 0$ for any $j \leq i$. Similarly, for any i such that $s[i] = 1$, then $s[j] = 1$ for any $j \geq i$. This property is equivalent to the first 2 assumptions. The egg count is simulated by the chances d, and the action of dropping an egg is the same as querying the string at a specific position.

This problem is related to binary search, as for the case of $e \geq \log_2(n)$, it can be solved exactly in $O(\log_2(n))$ time. The interesting case is when there is the limiting constraint.

4 Tight Bound for $e = 2$

Consider that every algorithm makes the following starter queries

$$s[p_1], s[p_2], \ldots, s[p_k]$$

where $p_i < p_j$ for $i < j$. This is because, until the final query is $s[p_k] = 1$, querying smaller positions does not provide any information, as we know the structure of the string is m 0s followed by $n - m$ 1s. Thus, to consider the case $s_n = 0^n 1^{n-n} = 0^n$, it must be the case that $p_k = n$.

Let the distance between each of these positions be denoted as

$$d_1 = p_1, \qquad d_i = p_i - p_{i-1} \quad \text{for } i \geq 2$$

Upon seeing $s[p_i] = 1$, consider the information that the algorithm has

$$s = b_1 \dots b_{p_{i-1}} b_{p_{i-1}+1} \dots b_{p_i - 1} b_{p_i} \dots b_n = 0^{p_{i-1}} [b_{p_{i-1}+1} \dots b_{p_i - 1}] 1^{n - p_i}$$

Thus, in order to distinguish between s_m for $m \in M \coloneqq \{m \in \mathbb{Z} : p_{i-1} < m < p_i\}$, the algorithm must linearly search for each position in the set. This gives us

$$T_i = i + d_i$$

That is, the total number of queries is the number of queries before 1 is returned plus the distance between the last query and the second to last query. Since the goal of the algorithm is to minimize the worst case, it must be the case that $T_i = T_j$ for all i, j, so let $T = i + d_i$ for all i be the total query count of the algorithm. Rewriting it gives us $d_i = T - i$

Notice that we have

$$\sum_{i=1}^{k} d_i = p_1 + p_2 - p_1 + \ldots + p_k - p_{k-1} = p_k = n$$

So we have

$$n = \sum_{i=1}^{k} d_i = \sum_{i=1}^{k} T - i = kT - \frac{k(k+1)}{2}$$

And thus, we want to minimize T by

$$T = \frac{n}{k} + \frac{k+1}{2}$$

If we consider T as a continuous function of k, to minimize T by k, we consider

$$T' = -\frac{n}{k^2} + \frac{1}{2} = 0$$

Solving for k, we get

$$\frac{k^2}{2} = n \Rightarrow k = \sqrt{2n}$$

Thus giving us the lower bound for $d = 2$. This is in fact tight, as we follow the lower bound. We query at positions $i \cdot \sqrt{n}$ for $i = 1, 2, ..$ until we see 1, then linearly search the space in between. Analyzing the time complexity yields exactly

$$\frac{n}{\sqrt{n}} + \sqrt{n} = 2\sqrt{n} = O(\sqrt{n})$$

So this is tight.

5 Tight Bound for $e = o(\log n)$

This problem can be generalized to any arbitrary e eggs, and the goal is to show that the tight bound is in fact $O(n^{1/e})$.

Let $n \in \mathbb{N}$, $[n]$ denoting the set $\{i \in \mathbb{Z} : 0 \leq i \leq n\}$, e be the amount of eggs, and A be the algorithm to solve the problem.

Game $\text{ED}_{n,e}$

procedure Initialize
$m \xleftarrow{\$} [n], E \leftarrow e$
$s \leftarrow 0^m || 1^{n-m}$

procedure Query(p)
If $s[p] == 1$ then $E \leftarrow E - 1$
Return $s[p]$

procedure Finalize(m')
Return $m' = m$

Consider the sequence of queries A makes in order to solve the problem.

$$P = p_{1,1}, \ldots, p_{1,k_1}, p_{2,1}, \ldots, p_{2,k_2}, \ldots, p_{e,k_e}$$

where $p_{d,k}$ denotes the k-th query the algorithm makes after breaking an egg and using the d-th egg. Thus, $s[p_{d,k_d}] = 1$ for all d and 0 otherwise. If A finalizes while having $d < e$, then $k_{e'} = 0$ for all $e' > d$.

Consider splitting the sequence into different stages, as in, for some d,

$$Q_d = p_{d,1}, \ldots, p_{d,k_d}$$

This captures the queries of A when using the d-th egg. Note that within each stages, the query position must be strictly increasing. Only with the last query of each stage does A drop an egg, so querying smaller positions before that does not yield any information, as for all p such that $s[p] = 0$, $s[p'] = 0$ for all $p' \leq p$. Using this, we can see that the total number of queries A makes is

$$T = \sum_{i=1}^{e} k_i$$

This is indeed the number of queries A makes within each stage. While it is nice to look at, each k_i can depend on the previous stages, which rather complicates things. Thus, we want to contain the relationship between these stages. Notice that upon breaking an egg, the game reduces to a strictly smaller instance of ED, where the relationship is rather simple.

Lemma 1. Upon breaking an egg, the query complexity of the remaining problem is the same as an instance of a strictly smaller problem. In fact, this instance only depends on the distance between the last and second to last query.

Proof. Consider any instance of a problem $\text{ED}_{N,E}$, and any algorithm A has made the sequence of queries $p_{1,1}, \ldots, p_{d,k_d}$ for some d, so that $s[p_{d,k_d}] = 1$, that is, A just broke an egg. For convenience, let $n = p_{d,k_d} - p_{d,k_d-1}$ be the distance between the last and second to last query. Consider a separate instance, $\text{ED}_{n,e-d}$. The number of remaining eggs are the same, and the distances are also the same. It should be clear that these problems are now exactly as difficult.

To be precise, given any algorithm A solving one instance, we can reduce it to solve the other instance. That is, let A make queries P to solve $\text{ED}_{n,e-d}$. We thus only need to increment each query by p_{d,k_d-1} to solve the bigger instance of the problem. Similarly, let A make queries P' to solve the bigger instance. Simply subtract out p_{d,k_d-1} to solve $\text{ED}_{n,e-d}$.

With that, we showed how, at every stage transition, the problem reduces to a strictly smaller subproblem, and the only dependence between them is the distance between the last two queries. Now, we want the distance between queries to be regular.

Lemma 2. Within each stage, the distances between the queries are all equal.

Proof. This one is rather straight forward from the previous lemma. It states that the distance between the last two queries matter. If it's small, then the reduced problem size is smaller, and vice versa. The goal of A is to minimize the worst case, so it would want to equalize all of the distances to reduce the extreme of the worst case.

With those two lemmas, let's rewrite the total query cost. Within each stage, the total number of queries is simply the size of the instance over the distance. Additionally, the size of the instance is exactly the distance between queries from the previous instance, giving us the total query cost formula as

$$T = \frac{d_0}{d_1} + \frac{d_1}{d_2} + \cdots + \frac{d_{e-1}}{d_e}, \qquad 1 \le d_i \le d_{i-1},\ 0 \le i \le e$$

Note that $d_0 = n$, since that is the size of our original problem instance, and $d_e = 1$, otherwise, it would skip over some floors and not be exact. Thus, we want to minimize this function according to the constraints.

We can find the minimum by using the AM-GM inequality, which states for any list of n nonnegative real numbers $x_1, x_2, ..., x_n$

$$\frac{\sum_{i=1}^{n} x_i}{n} \ge \sqrt[n]{\prod_{i=1}^{n} x_i} \Longrightarrow \sum_{i=1}^{n} x_i \ge n \cdot \sqrt[n]{\prod_{i=1}^{n} x_i}$$

where equality holds if and only if $x_i = x_j$ for all i, j. We simply let $x_i = d_{i-1}/d_i$ and $n = e$, which gives us the equation for the total query cost

$$T = \frac{d_0}{d_1} + \frac{d_1}{d_2} + \cdots + \frac{d_{e-1}}{d_e} = \sum_{i=1}^{e} x_i$$

Since equality of all terms implies the minimum sum, consider $d_i = n^{1-i/e}$

$$x_i = \frac{d_{i-1}}{d_i} = \frac{n^{1-(i-1)/e}}{n^{1-i/e}} = \frac{n^{i/e}}{n^{(i-1)/e}} = \left(\frac{n^i}{n^{i-1}}\right)^{1/e} = n^{1/e}$$

Thus, we simply get, since $e = o(\log n)$

$$T = \sum_{i=1}^{e} x_i = e \cdot n^{1/e} = O(n^{1/e})$$

So we conclude the lower bound is correct. Following this construction, we can simply follow the same steps and get a concrete algorithm for this as well. Using the notation detailed above, the optimal algorithm simply queries

$$p_{d,k_i} = p_{d-1,k_d-1} + k_i \cdot n^{1-d/e}$$

using the d-th egg on the k_i-th step.

6 Additive Error Approximation

We consider the same problem setup, but we also accept if the algorithm is the answer is within the range $[m \pm k]$, where we refer to k as the error.

6.1 Algorithm for Additive Error

An algorithm that can solve this is as follows: divide n floors into k-size buckets, and we consider these buckets as if each of them is one floor. We consider the representative floor to be the lowest floor in each bucket to query, and proceed with the optimal algorithm above. Once we find the bucket that m is in, we simply guess the lowest floor, which must be at most k floors away, since both of them must be in the same bucket.

As we can see, the runtime is simply $O((n/k)^{1/d})$, as we're dividing n floors into k-size buckets, and querying only 1 representative floor within each bucket.

6.2 Lower Bound for Additive Error

Consider any algorithm that approximates m by some error k, outputting m' with some runtime T. With that, we have

$$|m' - m| \leq k \implies m' - k \leq m \leq m' + k$$

We can upgrade this to an exact algorithm by, with one extra egg, linearly searching through the last $2k$ floors. Thus, with any approximation algorithm with error k with runtime T, we extract an exact algorithm with runtime $T + 2k$. By our lower bound above, it must be the case that

$$O(T + 2k) \geq O\left(n^{1/e}\right)$$

If we assume that T is optimal, we have the inequality for lifting an approximate algorithm to an exact one as

$$O\left(\left(\frac{n}{k}\right)^{1/(e-1)}\right) + k \geq O(T + 2k) \geq O\left(n^{1/e}\right)$$

We can simply let k be expressed as n^c for $c = \log_n(k)$, since k is a strictly positive integer at most n, giving us that $0 \leq c \leq 1$. This way, we get

$$O\left(\left(\frac{n}{n^c}\right)^{(e-1)}\right) + n^c \geq O(T + 2n^c) \geq O\left(n^{1/e}\right)$$

We can see this is indeed tight for k if we solve for c to minimize the left quantity. We let

$$n^c = \left(\frac{n}{n^c}\right)^{1/(e-1)} = \frac{n^{1/(e-1)}}{n^{c/(e-1)}} \Longrightarrow c = \frac{1}{e}$$

Which makes intuitive sense. This last step of linearly searching through the gap is exactly the last stage for the exact algorithm.

7 Quantum Algorithm

We include a brief comment on a quantum algorithm that can solve this problem. We can frame this problem as a Elitzur-Vaidman Bomb problem [3]. That is, every drop can be considered as a standard basis measurement, where if the measurement is 0, then the egg is broken, otherwise, the egg can be reused.

Applying the Elitzur-Vaidman Bomb Detector algorithm simply allows us to detect the bomb and never trigger it. That is, we can always detect, with very high probability, whether an egg will break or not on a certain floor without actually breaking the egg. This immediately gives us the power to use an unlimited amount of eggs, which reduces this problem back to binary search, which yields a runtime of $O(\log(n))$. While there is some overhead to query mutiple times on the same floor to implement the Elitzur-Vaidman Bomb Detector algorithm, it is constant in terms of n, the input size. With that, there is a clear quantum advantage for this problem.

References

[1] DataGenetics. *The Two Egg Problem*. Last accessed 9 December 2025. DataGenetics, 2012. URL: http://www.datagenetics.com/blog/july22012/index.html.

[2] GeeksforGeeks. *Egg Dropping Puzzle*. Last accessed 9 December 2025. GeeksforGeeks, 2025. URL: https://www.geeksforgeeks.org/dsa/egg-dropping-puzzle-dp-11/.

[3] Qipeng Liu. *CSE 190 Notes 4*. 2024. URL: https://drive.google.com/file/d/1sSyAYi-CAzacQ3oVoy0mY7pgPOaVop27/view.

[4] Marcin Moskala. *How to solve the Google recruiters' puzzle about throwing eggs from a building*. Last accessed 9 December 2025. freeCodeCamp, 2017. URL: https://www.freecodecamp.org/news/how-to-solve-the-google-recruiters-puzzle-about-throwing-eggs-from-a-building-de6e7ef1755d.