Lecture 22, Tues April 11: Grover

The next quantum algorithm we'll cover is…

Grover's Algorithm

which was discovered in 1995, shortly after Shor's algorithm.

Both Grover and Shor were working at Bell Labs at the time.

Grover's algorithm gives a smaller speedup than Shor's (quadratic rather than exponential), but for a much wider range of problems. Just like with all the other quantum algorithms we've seen, it's easiest to think of Grover's algorithm in terms of a black box:

Given an oracle function f: $\{1, \dots, N\} \rightarrow \{0, 1\}$.

We'd like to answer two questions:

- Is there an x such that $f(x) = 1$?
- If there is, what is such an x?

The basic problem that Grover's Algorithm addresses is unordered search.

a.k.a looking through a list of bits for a 1 bit.

Classically, we'd need a linear number of queries, $\Omega(N)$, to solve this problem deterministically.

Why? Simply because, if we want to know for certain whether there's a treasure hidden in one of n boxes, then even after opening N–1 boxes and finding them empty, we still need to open the N^{th} box!

If the treasure is guaranteed to be in *some* box, then finding it takes $\sim \frac{N}{2}$ queries on average, which is still linear in N.

Grover's algorithm solves both problems using only $O(\sqrt{N})$ quantum queries to the function f.

This might seem unreasonable---but as we'll see, it's quite similar to how the Elitzur-Vaidman Bomb worked.

The number of qubits needed to run Grover's algorithm is very low, $O(\log N)$, and the number of gates required is also reasonable, $O(\sqrt{N} \log N)$.

However, for Grover's algorithm to work, we do need to assume that we have quantum access to the function f, in such a way that we can apply the unitary transformation $|x, a\rangle \rightarrow |x, a \oplus f(x)\rangle$. This wasn't important in Shor's Algorithm because we only made one query and then discarded the result.

There are two main example applications to keep in mind with Grover's algorithm.

The first application is solving combinatorial search and optimization problems, such as NP-complete problems. Here, we think of $N = 2^n$ as being exponentially large, and we think of each candidate solution $x \in \{0,1\}^n$ as an n-bit string. We then set, for example,

$f(x) = \varphi(x)$,

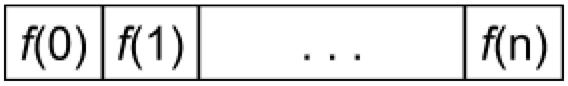
where φ is an instance of Satisfiability or some other **NP**-complete problem.

Then Grover's algorithm can solve the problem in $O(2^{n/2}\, \mathrm{poly}(n))$ time: namely, $\sqrt{N} = 2^{n/2}$ queries to f, and poly(n) time to implement each query (say, by checking whether a given x satisfies φ).

This is a speedup for **NP**-complete problems---but at most a quadratic one, and also only *conjectural*, because of course we can't even rule out the possibility of **P**=**NP**, which would annihilate this sort of speedup.

For an **NP**-complete problem like CircuitSAT, we can be pretty confident that the Grover speedup is real, because no one has found any classical algorithm that's even slightly better than brute force. On the other hand, for more "structured" **NP**-complete problems, we *do* know exponential-time algorithms that are faster than brute force: for example, 3SAT is solvable in about $O(1.3^n)$ time. So then the question becomes a subtle one, of whether Grover's algorithm can be *combined* with the best classical tricks that we know, to achieve a polynomial speedup even compared to a classical computer that uses the same tricks. For many **NP**-complete problems, the answer seems to be yes, but it need not be yes for all of them.

The second example application of Grover's algorithm to keep in mind---Grover's original application---is searching an actual physical database.



Say you have a database of personnel records, and you want to find a person who matches various conditions (hair color, hometown, etc).

You can set $f(x) = 1$ if person x meets the criteria
0 otherwise

Then Grover's algorithm can search for an x such that $f(x) = 1$ in $O(\sqrt{N})$ steps.

Some people have questioned the practicality of using Grover's algorithm to search a physical database, because the database needs to support "superposed queries": that is, you need to be able to query many records in superposition and get back a superposition of answers.

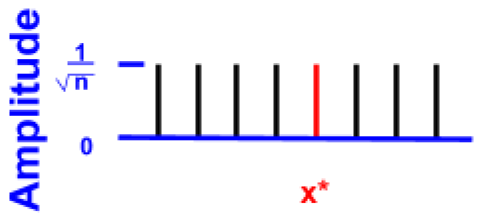
A memory that would support these kinds of queries is called a "quantum RAM." Building one is a whole additional technological problem, beyond building a quantum computer itself. And it remains unclear whether people will be able to build quantum RAMs without n active, parallel computing elements---which, if you had them, would remove the need to run Grover's algorithm. In this lecture, though, we'll treat such things as "mere engineering difficulties"!

Anyway, one big advantage of Grover's algorithm as applied to actual physical databases is that there the quantum speedup is <u>provable</u>: it doesn't rely on any unproved computational hardness assumptions.

OK, so without further ado, <u>how does Grover's algorithm work?</u>

For simplicity, let's assume that a solution exists and is unique. We call this the <u>marked item</u>: it's the unique x^* such that $f(x^*) = 1$

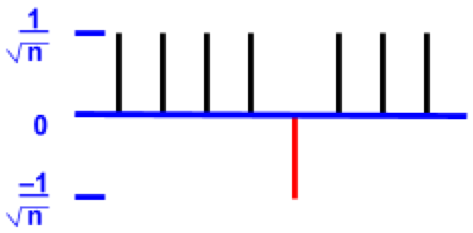
We'll also assume for simplicity that $N = 2^n$ is a power of 2, which will allow us to do our favorite trick: start by Hadamarding n qubits.



Doing so brings the initially all-0 state to a uniform superposition:

$$|00\ldots0\rangle \to (1/\sqrt{N}) \sum_{x=1}^{N} |x\rangle$$

As shown, all possible x values have the same amplitudes.

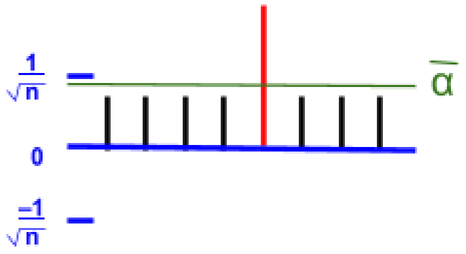


Then we query with U_f, a unitary transformation that flips the amplitude of the marked item: $U_f|x\rangle = (-1)^{f(x)}|x\rangle$.

As we've already seen in this course, if we can apply $|x, a\rangle \to |x, a \oplus f(x)\rangle$, then we can also apply the phase oracle $|x\rangle \to (-1)^{f(x)}|x\rangle$. Phase oracles are more convenient for the purposes of Grover's algorithm.

Next we apply a unitary matrix D (shown below), the so-called "Grover diffusion operator," which has the effect of flipping all N amplitudes about the mean amplitude.

$$\alpha_x \to 2\overline{\alpha} - \alpha_x \text{ where } \overline{\alpha} = \frac{1}{N}\sum_{x=1}^{N} \alpha_x$$

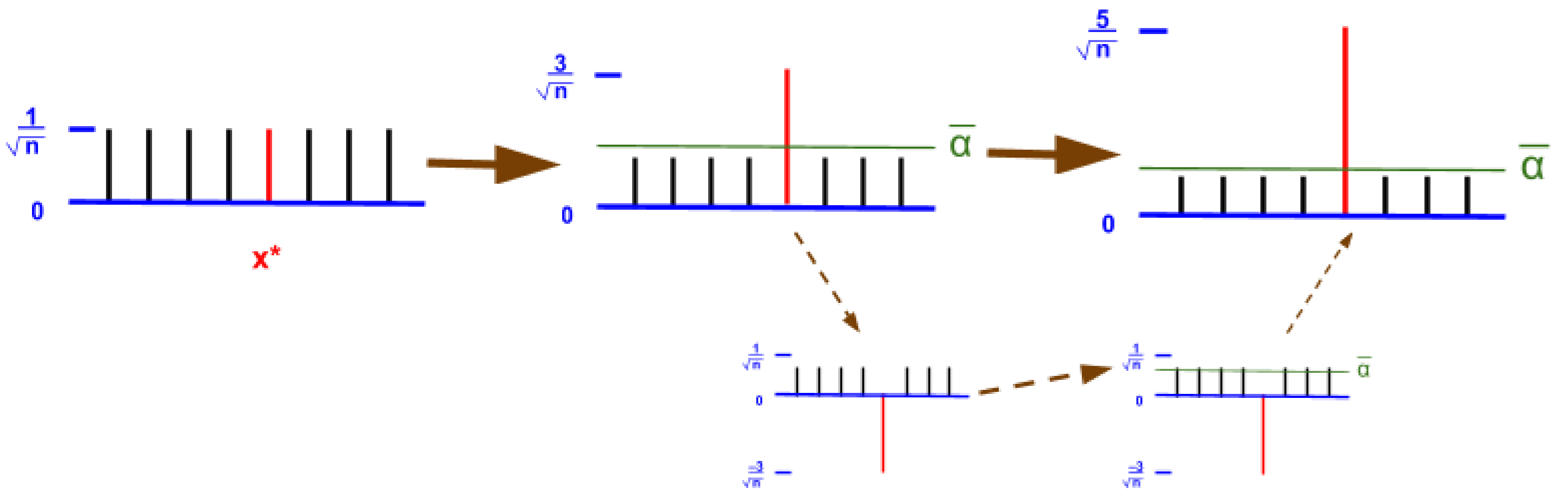


$$D = \begin{bmatrix} \frac{2}{N}-1 & \frac{2}{N} & \cdots & \frac{2}{N} \\ \frac{2}{N} & \frac{2}{N}-1 & \cdots & \frac{2}{N} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{2}{N} & \frac{2}{N} & \cdots & \frac{2}{N}-1 \end{bmatrix}$$

So why does applying D help us?

Well, let's look at what's happened after a single Grover iteration. We've managed to increase the amplitude of the marked item, to roughly $3/\sqrt{N}$, and decrease the amplitudes of all the other items.

Now we keep repeating: another U_f, then another D, then another U_f, then another D, and so on. If we run those steps repeatedly, we can increase the amplitude of the marked item further as pictured below.



As an approximation, we can say that when the number of queries is small, the repetition increases the marked item's amplitude as

$$\frac{1}{\sqrt{N}}, \frac{3}{\sqrt{N}}, \frac{5}{\sqrt{N}}, \frac{7}{\sqrt{N}}, \ldots$$

Notice that it would take $O(\sqrt{N})$ steps for this series to reach $\frac{\sqrt{N}}{\sqrt{N}} = 1$.

This is a quadratic speedup. Classically we'd need about N queries to find the answer, because after t queries we'd have a probability of $\frac{t}{N}$ of having found the marked item. Meanwhile, if we measure after t queries in Grover's algorithm, we find the marked item with probability of order $\left(\frac{2t}{\sqrt{N}}\right)^2 \sim \frac{4t^2}{N}$.

This picture isn't exactly right, though, because we ignored some details. Over time the mean gets smaller, so the increase in the marked item's amplitude slows down.

Which makes sense, because otherwise the amplitude would continue increasing past 1!

We'll see exactly what happens shortly.

First, though, a natural question to ask about Grover's algorithm is "Why should it take $\sqrt{N}$ steps? Why not $\sqrt[3]{N}$ or $\log N$?"

We see here that, in some sense, the ultimate source of the $\sqrt{N}$ is the fact that amplitudes are the square roots of probabilities. Instead of adding $1/N$ probability to the marked item with each query, quantum mechanics lets us add $1/\sqrt{N}$ amplitude, resulting in quadratically faster convergence.

Of course, if we want to use Grover's algorithm in practice, then in addition to bounding the number of queries by $O(\sqrt{N})$, we'll *also* need to find a small quantum circuit to implement the Grover diffusion operator D.

Say we want to implement D on an n-qubit state ($N = 2^n$). It's easiest if we look at what D does in the Hadamard basis. What *does* it do in that basis?

Well, the β^th amplitude would be $\beta_s = \frac{1}{\sqrt{N}} \sum_{x=0}^{N-1} (-1)^{s \cdot x} \alpha_x$

We've seen previously that this is the result of switching to the Hadamard basis.

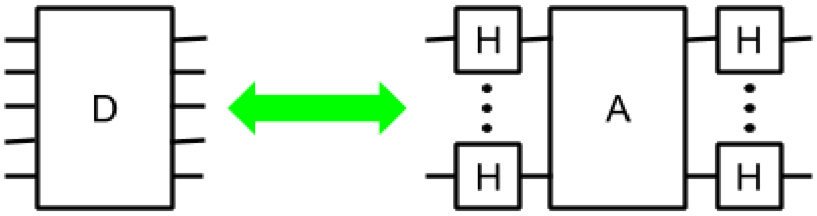
So what happens to these β_s's?

The first one plays a special role: if $s = 0$, we have something proportional to the average, which is good because our goal was to invert about the average. The other s values play no particular role in Grover's algorithm.

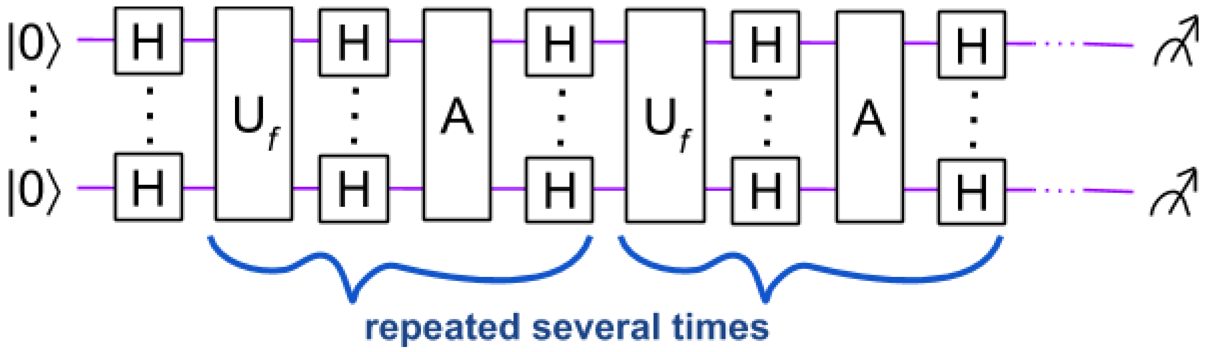
So in the Hadamard basis, if you think about it, what we want is to perform the diagonal matrix A on the right. This A is easy to implement as a quantum circuit, by using some ancilla qubits to check whether the input is all 0's, inverting the phase if not, and finally uncomputing garbage (details left as an exercise).

$$A = \begin{bmatrix} 1 & & & \\ & -1 & & \\ & & -1 & \\ & & & -1 \end{bmatrix}$$

Again, we claim that the A transformation, conjugated by Hadamard gates on either side, just implements the Grover diffusion transform D. If you don't believe this, you can verify it by an explicit calculation.



Our final circuit for Grover's Algorithm (with ~ $\sqrt{N} \log N$ gates and ~ $\sqrt{N}$ queries to f) is drawn below.



Now let's analyze Grover's algorithm more carefully, and actually prove that it works.
e.g. we haven't yet shown that O($\sqrt{N}$) queries is enough to take us to Pr(success) ≈ 1.

Let's call the initial state $|\Psi\rangle = \frac{1}{\sqrt{N}} \sum_{x=0}^{N-1} |x\rangle$

Somewhere in N-dimensional space is the basis state corresponding to the marked item we're looking for: $|x^*\rangle$.

<u>What is ⟨Ψ|x*⟩?</u>

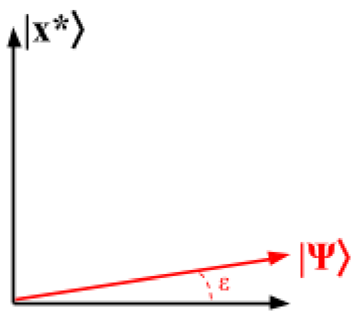
It's $\frac{1}{\sqrt{N}}$. So $|\Psi\rangle$ and $|x^*\rangle$ are not quite orthogonal, but *almost*.

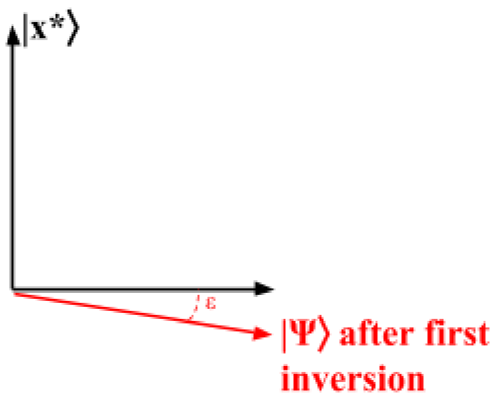
Now, even though $|\Psi\rangle$ and $|x^*\rangle$ are not orthogonal, these two vectors span a two-dimensional subspace. A crucial insight about Grover's algorithm is that <u>it operates entirely within this subspace</u>. Why? Simply because we start in the subspace, and then neither the queries nor the Grover diffusion operations can ever cause us to leave it!

This means that we can visualize everything Grover's algorithm is doing by just drawing a picture in the plane.

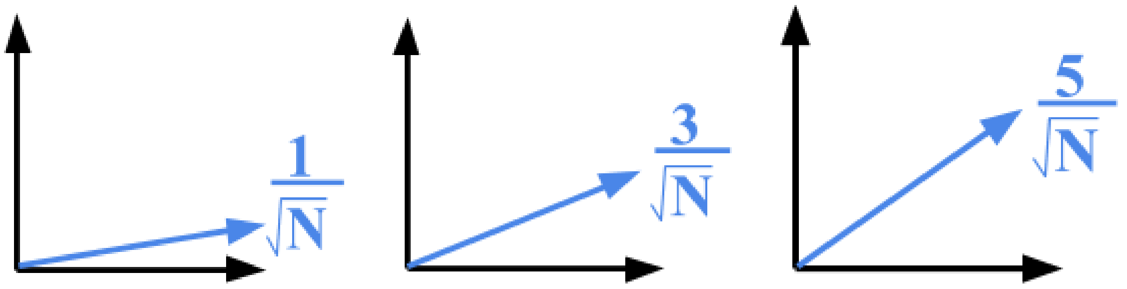
We saw how the algorithm alternates between two types of operations:

- Reflecting our state about $|\Psi\rangle$ (the diffusion transform D)
- Inverting the component of our state that points in the $|x^*\rangle$ direction (a query U_f)





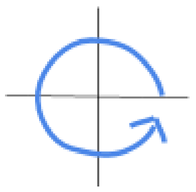
So the initial angle with the horizontal is $\theta = \arcsin(\frac{1}{\sqrt{N}}) \approx \frac{1}{\sqrt{N}}$



And then we rotate by $\frac{2}{\sqrt{N}}$ at each iteration. This means that we'll get super close to $|x^*\rangle$, and have a high probability of observing x^* when we measure, after about $\frac{\pi}{4}\sqrt{N}$ iterations--something that you can directly see from the geometric picture. We might not get *exactly* to 1, if the step size of the rotations causes us to overshoot the vertical direction slightly, but at any rate we'll get close.

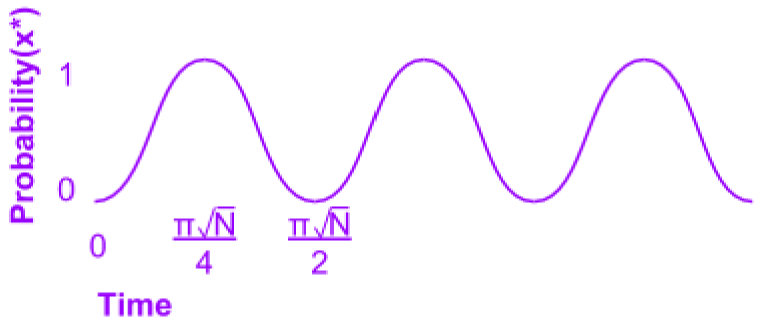
There are several interesting things about this picture...
<u>What happens if we run Grover's algorithm for too long?</u>

Well it has a property that almost no classical algorithm has: it starts getting worse.



Grover's algorithm has been compared to baking a souffle:
Once risen, it must be taken out of oven or it'll get short again.
On the other hand, Grover's algorithm has at least one property not shared by souffles: namely, if you "leave it in the oven" for even longer, it rises a second time, then goes down a second time, and so on forever!

Graphing the success probability over time produces a sinusoidal curve, since the probability is just the squared projection of the current quantum state onto the y-axis (i.e., $\sin^2\Theta$):



Grover's algorithm can also put you into an interesting dilemma:

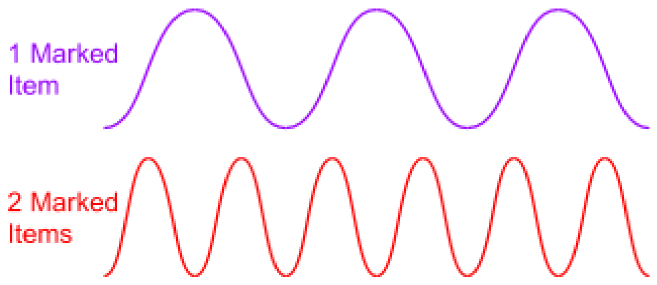
Suppose you've run the algorithm for a given number of iterations, fewer than $\frac{\pi}{4}\sqrt{N}$. Then you could measure right now, and take your chances with whether you'll observe the solution, or you could let it run longer to boost your chances. If you measure right now and *don't* see the solution, then you need to start over from the very beginning.

This could make for an interesting science-fiction story: the heroes need to break a cryptographic code to beat villains, so they run Grover's algorithm over all the possible decryption keys. They've run it for a day and gotten up to 45% probability of observing the solution, but now the villains have entered their compound and are closing in on them. So, do they measure now, or do they let the algorithm run a bit more?

<u>Could you get a solution faster by doing many runs, each with measurement after a shorter amount of time?</u>

It depends on the desired level of confidence in getting the right answer after a set amount of time. If your goal is just to minimize the *average* number of queries until you learn the answer, then it turns out to be optimal to stop some amount of time before you're able to get a guaranteed answer (details left as an exercise).

For simplicity, everything we said above assumed exactly one marked item. What happens if there are more? Simple: the entire cycle happens faster. The more items, the faster the cycle.



In particular, if *k* items out of *N* are marked, then Grover's Algorithm peaks at $\frac{\pi}{4}\sqrt{\frac{N}{k}}$ queries.

One of the first questions people asked about Grover's algorithm was, "but what if the number of marked items isn't known?"

You can sort of see the danger. It's possible to run Grover's algorithm the right number of times to hit the peak when there's a single marked item, but that might result in a trough if the number of marked items is larger.

The most basic way to solve this problem is simply to run the algorithm for a *random* number of iterations, say between 0 and $2\sqrt{N}$. If we do this, then most of the time, we expect to end up somewhere around the middle of a sinusoid, neither at a trough nor a peak, where we have a constant probability (say, 40% or 60%) of observing a solution if we measure. This is perfectly sufficient from an algorithmic standpoint, since it means that we only need to repeat the algorithm O(1) times on average.

This gives us an upper bound of $O(\sqrt{N})$ queries to find a marked item with high probability, regardless of how many there are (assuming there's at least one).

While we weren't explicit about this point before, given a candidate solution x output by Grover's algorithm, we can simply evaluate $f(x)$ classically to check whether x is really a marked item.

What happens if we run Grover's algorithm, but the database turns out to have no marked items?

When we query f: nothing happens

When we do the diffusion transform: nothing happens

So, the state just remains a uniform superposition over all N items for the entire duration of the algorithm. This means that when we measure, we just get a random item. Then we can check that item and see that it isn't marked.

How can we be certain that there no marked items?

This is the question that arises in the decision version of Grover's algorithm. In fact, no matter how many times we run Grover's algorithm, we never become *100% sure* that there are no marked items, since we could've just gotten unlucky and failed to find the items every time. However, because after O(1) repetitions, the algorithm has as a high a probability as we like (say, >99.99%) of finding a marked item assuming that there's at least one of them, if after that time it *hasn't* found a marked item, then we can deduce that there almost certainly weren't any. This again requires only $O(\sqrt{N})$ queries.

We've said that if there are k marked items, then we find one of them in $O(\sqrt{N})$ queries without knowing k. But in fact we can do even better than that, and find a marked item in only $O(\sqrt{N/k})$ queries, again without knowing k: the same performance as if we *did* know k. How?

Assume for simplicity that N is a power of 2.

Then first, we guess that almost all items are marked: we do a single query, then measure. If we find a marked item, great.

If not, next we guess that $\frac{N}{2}$ items are marked, and run Grover's algorithm with $k=N/2$. If we find a marked item, great.

Next we run Grover's algorithm with $k=N/4$, then $k=N/8$, and so on, repeating halving our guess for the number of marked items, until either we've found a marked item or we've searched unsuccessfully with $k=1$.

This method wastes some queries on "wrong" values of k. But crucially, because the number of queries is increasing exponentially, the number of wasted queries is only a constant factor greater than the number of queries used in the final iteration: the one that guesses an approximately correct value of k. Details of the analysis are left as an exercise.

Let's end by mentioning a different way to handle the case of multiple marked items. This way achieves essentially the same performance using a purely classical trick.

Again suppose we have N items, k of which are marked. <u>We want to reduce to the case of just a single marked item.</u>

How do we do that? Simple: we pick $\frac{N}{k}$ items uniformly at random, and then run Grover's algorithm on that subset only.

The number of marked items that we'll catch has a Poisson distribution. And one can calculate the probability of catching exactly one marked item in our sample as $\sim \frac{1}{e}$. So, we search that subset of $\frac{N}{k}$ items, using Grover's algorithm for the single marked item case (which uses O($\sqrt{N/k}$) queries). If we don't find a marked item, we can try again with a new random subset.

Exercise for the reader: Show that, if there are k marked items and we want to find *all* of them, we can do that using O($\sqrt{Nk}$) queries.