Sean Carroll, seancarroll@gmail.com

125c, Lecture Sixteen : 5/24/17

Let's be a bit more systematic about

Computational Complexity Theory.

We're not talking about complexity in the sense of "complex systems," but rather the complexity of problems – well-defined problems that could in principle be solved by a computer. By "complexity" we really mean resources – in particular, resources of time (How many steps does it take to solve the problem?) and space (How much memory/storage is required?)

Mostly we'll be worrying about time.

We don't care about <u>specific</u> problems as about <u>kinds</u> of problems. In particular, there is a number n representing the "length of the input" for our problem. If we're sorting or searching a list, n is just the number of entries in the list. If we're asking for the shortest route between a set of cities, n = the number of cities; if you're factoring a large number, n = number of digits in the number.

Also: many interesting computational problems are <u>hard</u>, and n is often very large. (Search for "Lady Gaga" in the list of all Google search queries over the last ten years.)

So we care less about the <u>exact</u> number of steps, and more about how that number <u>scales</u> as $n \to \infty$ ("asymptotic behavior"). Therefore use "Big O Notation":

$$f(n) \in O(g(n)) \quad \text{iff} \quad \exists \ \lambda > 0$$

$$\text{s.t.} \quad \lim_{n \to \infty} |f(n)| \leq \lambda g(n).$$

Roughly, "keep the leading behavior at large n." Examples:

$f(n) = 3n^2 + 4n - 5$: $O(n^2)$

$f(n) = n^{1000} + 2^n$: $O(2^n)$

$f(n) = \sqrt{n} + \log n$: $O(n^{1/2})$.

$f(n) = \log(n^3)$: $O(\log n)$.

Some common scaling behaviors:

$$O(1) < O(\log n) < O(n^{\alpha<1}) < O(n)$$
$$< O(n \log n) < O(n^{\alpha>1}) < O(a^n) < O(n!).$$

A <u>complexity class</u> is a set of problems whose time to solve them scale in the same way. The field began by studying <u>classical</u> computation, where we typically imagine our calculation is being done on a "deterministic Turing machine" — roughly, a computer manipulating classical bits via fixed, deterministic rules. Famous complexity classes in this context:

P = problems solvable in polynomial time, $O(n^{\alpha>1})$.

EXP = problems solvable in exponential time, $O(a^n)$.

R = problems solvable in finite time.

} "on a deterministic Turing machine"

[There are also complexity classes that refer to spatial resources e.g. PSPACE = problems solvable with polynomial memory.]

Examples:

P:
- Is an integer x prime?
- What is the greatest common divisor of two integers x & y?

EXP:
- Chess. (Given a configuration on an $n \times n$ board, who wins?)

~~R~~: (i.e. not solvable in finite time.)
- Halting problem. Given a computer program and some input, will it eventually halt?

Roughly, problems in P are considered "easy", while those in EXP (but not in P) are considered "hard."
(e^n is much larger than n^α when n gets sufficiently large.)

Another famous class is

NP : Solvable in polynomial time on a <u>nondeterministic</u> Turing machine.

A nondeterministic Turing machine is one that can make random "lucky" guesses.

This is a silly definition of a simple concept. Really, NP problems are those for which it might be hard (exponential) to <u>solve</u>, but for which it's easy (polynomial) to <u>check</u> any purported solution. Examples:

- Traveling Salesman Problem: what is the shortest route connecting a set of n cities and distances between them?
- Factoring n-digit integers. Clearly NP; we don't know if it's in P (but suspect not).

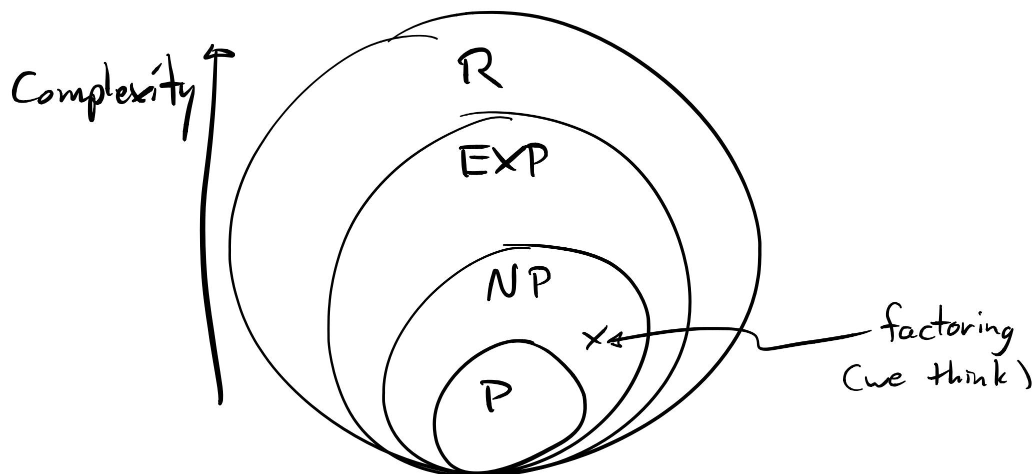
Conjecture all sensible people believe, but nobody has yet proven:

$$P \neq NP.$$

This is a "Millenium Problem" of the Clay Mathematics Institute; if you prove (or disprove) it, you win \$1,000,000.

This formalizes the idea that there are some problems that are hard to solve, but easy to check answers.

So we believe: $P \subset NP \subset EXP \subset R$.



Quantum computers have different capabilities, so we need new complexity classes. One caveat: a quantum computer is somewhat analogous to a <u>probabilistic</u> (rather than deterministic) classical one. Probabilistic computers sometimes give the wrong results!

Actually not a problem. Let's say an algorithm makes an error with probability guaranteed to be $\leq p$, but we really want an error probability $\leq \epsilon$. Just run the algorithm M times, where

$$M \geq \log_p \epsilon \implies p^M \leq \epsilon.$$

So for probabilistic computers we don't care about P, but about

BPP: bounded-error probabilistic polynomial time (error probability guaranteed $\leq 1/3$.)

Clearly $P \subseteq BPP$, since a deterministic computer is a special case of a probabilistic one.

We might think there are problems in BPP that are <u>not</u> in P — i.e. that allowing for probabilistic techniques enables parametric speedup. But no problems we know are in BPP are also known to <u>not</u> be in P. For a while "Is the integer x prime?" was known to be in BPP and suspected not to be in P, but in 2002 a deterministic polynomial-time algorithm was found. Now people <u>suspect</u>

$$P = BPP.$$

(Such suspicions are not always right.) Plenty of unsolved problems in this game.

For quantum computers, therefore, the notion of an "easy" problem is formalized as

BQP: bounded-error quantum polynomial time.

Quantum computers can run classical algorithms, so it's immediate that

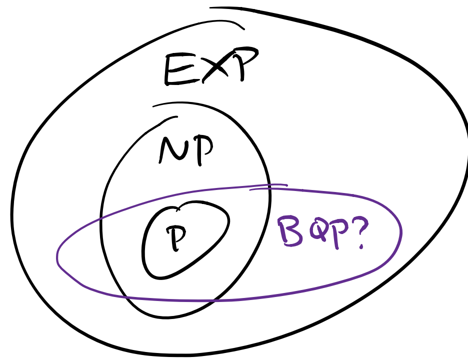
$$BPP \subseteq BQP \longrightarrow P \subseteq BQP.$$

Also true, but maybe you have to think about it, is that classical computers can simulate quantum computers, given exponential time (just numerically solve Schrödinger's eq.). Therefore $BQP \subseteq EXP$.

What we <u>don't</u> know includes:

- Is $P \neq BQP$? (Are there problems that are classically hard, but easy on a quantum computer?) (Suspect yes)
- Is $NP \not\subseteq BQP$? (Can quantum computers solve <u>any</u> NP problem in polynomial time?) (Suspect no.)

Thus:



Lots of work to be done.

A watershed moment in quantum computing was Peter Shor's discovery in 1997 of Shor's algorithm, which showed that a quantum computer can factor (decompose an integer into its prime factors) in polynomial time.

I.e.

$$\text{FACTOR} \in \text{BQP}.$$

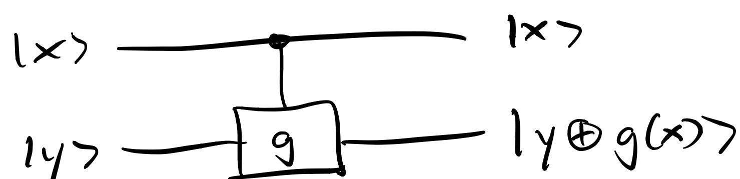
You might think this guarantees $P \neq BQP$, since factoring is classically hard. But we haven't actually proven that factoring is not in P. (Maybe the NSA has, but they haven't told us.)

We could describe Shor's algorithm, but it involves a lot of math that is fun, but not our main goal. Instead, let's study a quantum algorithm that provably gives some speedup over classical, even if both are in P: Grover's Algorithm for searching through an (unsorted!) list. We have a list of n elements $\{\alpha_i\}$ exactly one of which α_{i_*} satisfies some condition. I.e. there is a function

$$g(\alpha_i) = \begin{cases} 0, & i \neq i_* \\ 1, & i = i_* . \end{cases}$$

Classically this problem is $\mathcal{O}(n)$, since you just have to trudge through the list until you find $g(\alpha_i) = 1$. Grover provides speedup to $\mathcal{O}(n^{1/2})$.

For simplicity let's imagine $n = 2^m$, so we can represent our list elements using m qubits. We will have $g = 1$ on one of the m-qubit basis states, and $g = 0$ on the rest. As we did for Deutsch's algorithm, imagine a "controlled-g" gate:



where $|x\rangle$ is m qubits and $|y\rangle$ is one qubit.

In computer science this is called an "oracle" – it knows the answer to a question ("which input gives $g(x) = 1$?"), but won't tell us unless we ask correctly.

As we saw with Deutsch's problem, when we put $|-\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$ into the lower register, our C_g oracle gives

$$\frac{1}{\sqrt{2}}|x\rangle(|0\rangle - |1\rangle) \rightarrow (-1)^{g(x)}\frac{1}{\sqrt{2}}|x\rangle(|0\rangle - |1\rangle).$$

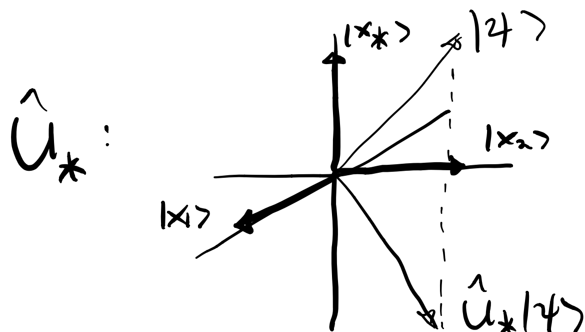
So we can think of this as effectively sending basis m-qubit states $|x\rangle \rightarrow (-1)^{g(x)}|x\rangle$.

If the state we're looking for is $|x_*\rangle$, this is equivalent to the unitary

$$\hat{U}_* = \mathbb{1} - 2|x_*\rangle\langle x_*|.$$

This is called "reflecting," since it will reflect an arbitrary $|\psi\rangle \in \mathbb{C}^{2m}$ around the plane perpendicular to $|x_*\rangle$.

$\hat{U}_*$:

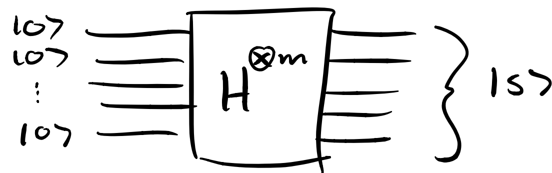


Grover's idea is this: start with a state that is an equal superposition of every basis state $|x\rangle$, then gradually increase the amplitude associated with $|x_*\rangle$, then iterate until the probability of measuring $|x_*\rangle$ is large ($>1/3$).

Thus, we begin with the m-qubit state (with $n = 2^m$)

$$|s\rangle = \frac{1}{\sqrt{n}} \sum_{x=000\cdots0}^{111\cdots1} |x\rangle .$$

In accord with our policy of using only computational basis vectors as inputs we can easily construct $|s\rangle$ by acting m Hadamards on input $|\underbrace{0000\cdots0}_{m}\rangle$:



If we "measure $|x\rangle$" (do a PVM onto the computational basis states), the probability of any specific outcome (including $|x_*\rangle$) would be

$$\left|\frac{1}{\sqrt{n}}\right|^2 = \frac{1}{n} \ll \frac{1}{3}.$$

We can amplify the correct answer $|x_*\rangle$ by combining the reflection $\hat{U}_*$ (by the oracle) with a reflection around the uniform state:

$$\hat{U}_s = 2|s\rangle\langle s| - \mathbb{1}.$$

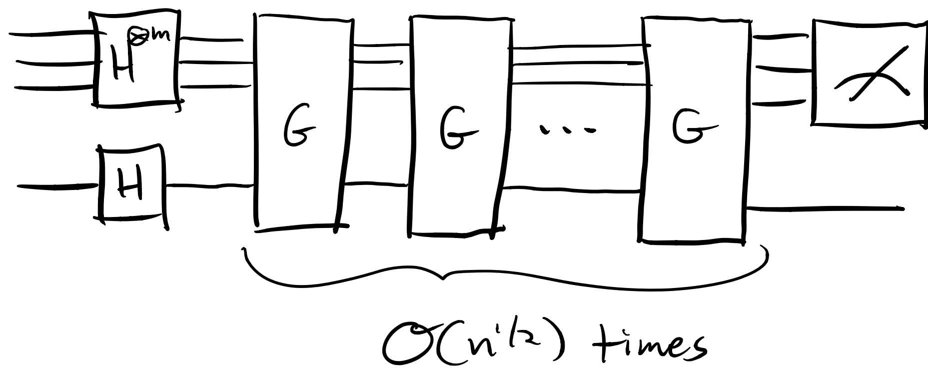
Note (or check for yourself) that this is equivalent to

$$\hat{U}_s = H^{\otimes m} \otimes \left(2|00\cdots0\rangle\langle 00\cdots0| - \mathbb{1}\right) \otimes H^{\otimes m}.$$

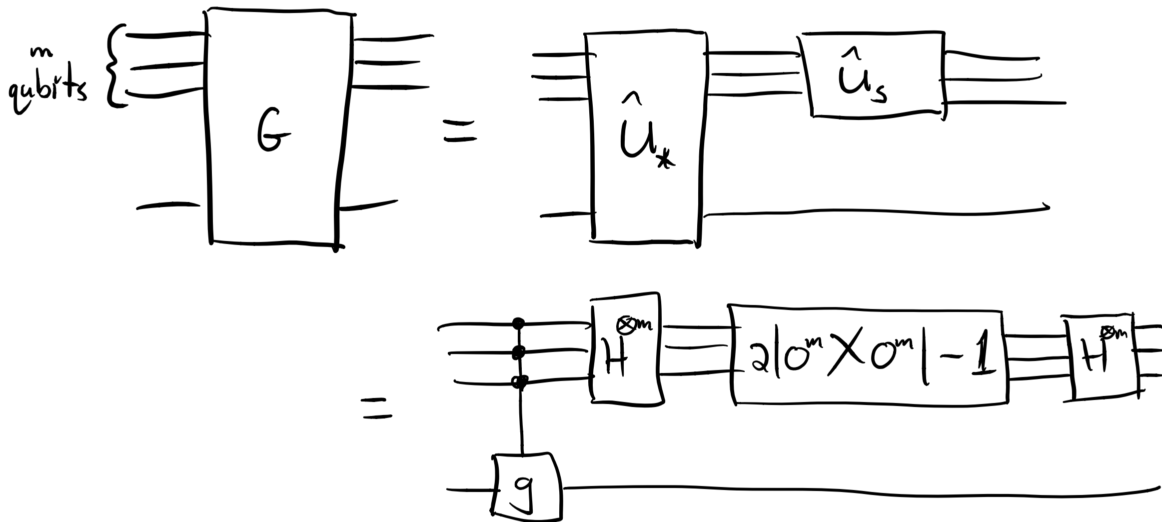
So starting with $|s\rangle$, Grover's algorithm is simply

1. Apply $\hat{U}_*$.
2. Apply $\hat{U}_s$.
3. Repeat $O(\sqrt{n})$ times.
4. Measure in computational basis.

As a circuit:



where



To see it in action, start with the first iteration.

- $|s\rangle = \frac{1}{\sqrt{n}} \sum_x |x\rangle$.
- $\hat{U}_* |s\rangle = (\mathbb{1} - 2|x_*\rangle\langle x_*|)|s\rangle$

$$= |s\rangle - 2\langle x_*|s\rangle |x_*\rangle$$

$$= |s\rangle - \frac{2}{\sqrt{n}}|x_*\rangle .$$

- $\hat{U}_s \cdot \hat{U}_* |s\rangle = (2|s\rangle\langle s| - \mathbb{1})(|s\rangle - \frac{2}{\sqrt{n}}|x_*\rangle)$

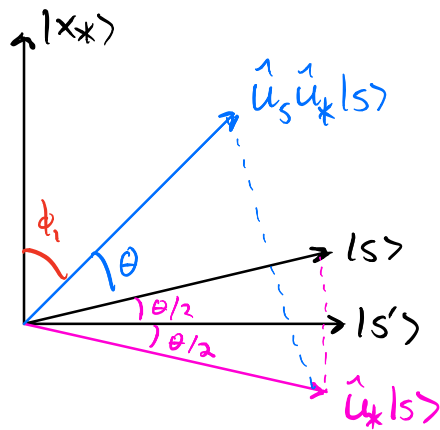
$$= 2|s\rangle - \frac{4}{n}|s\rangle - |s\rangle + \frac{2}{\sqrt{n}}|x_*\rangle$$

$$= \frac{n-4}{n}|s\rangle + \frac{2}{\sqrt{n}}|x_*\rangle$$

Not bad – we boosted the amplitude of $|x_*\rangle$ by a smidgen, and decreased all the others a bit. (Note that for $n=4$ we just got the answer exactly.)

There's a nice geometric interpretation of the algorithm. We have the ket we're looking for $|x_*\rangle$, and the symmetric ket $|s\rangle$. These two vectors in $\mathcal{H}$ determine a plane. We can construct a vector $|s'\rangle$ in this plane that is perpendicular to $|x_*\rangle$ just by subtracting $|x_*\rangle$ from $|s\rangle$ and normalizing:

$$|s'\rangle = \frac{1}{\sqrt{n-1}} \sum_{x \neq x_*} |x\rangle .$$



The angle $\theta/2$ between $|s\rangle$ & $|s'\rangle$ is

$$\sin\frac{\theta}{2} = \frac{1}{\sqrt{n}} .$$

$\hat{U}_*$ reflects in the plane perpendicular to $|x_*\rangle$; i.e. it reflects <u>around</u> $|s'\rangle$.

Likewise, $\hat{U}_s$ reflects around $|s\rangle$.

Together, $\hat{U}_*$ & $\hat{U}_s$ keep the vector in the $|x_*\rangle / |s\rangle$ plane, but rotate it closer to $|x_*\rangle$ by an angle

$$\theta = 2 \sin^{-1} \frac{1}{\sqrt{n}} .$$

Every iteration rotates by this same angle. After q iterations, the angle ϕ_q between $|x_*\rangle$ and $|s_q\rangle$ is

$$\phi_q = \frac{\pi}{2} - \frac{\theta}{2} - q\theta = \frac{\pi}{2} - (q+\tfrac{1}{2})\theta .$$

The probability of correctly measuring $|x_*\rangle$ is

$$\cos^2 \phi_q = \left[\cos\left(\frac{\pi}{2} - (q+\tfrac{1}{2})\theta\right)\right]^2 = \sin^2\left[(q+\tfrac{1}{2})\theta\right].$$

Highest probability is when $(q+\frac{1}{2})\theta \approx \pi/2$, or

$$q \approx \frac{\pi}{2\theta} \approx \frac{\pi}{4 \sin^{-1}(\frac{1}{\sqrt{n}})} \approx \frac{\pi}{4}\sqrt{n} = O(n^{1/2}).$$

So Grover's algorithm provides "quadratic speedup" for searches of unsorted lists when compared to classical computers,

$$\underset{\text{(classical)}}{O(n)} \longrightarrow \underset{\text{(quantum)}}{O(n^{1/2})}.$$

In fact it's been shown this is <u>optimal</u> – no quantum algorithm can search an unsorted list faster than $O(n^{1/2})$.

Amusing footnote: Scott Aaronson showed that hidden-variable theories allow for slightly faster speedup than Grover!

$$O(n^{1/3}).$$