Lecture 1 - Introduction

Goal: understand q. computing and q. algorithms using the tools of computer science (algorithms analysis, complexity theory, programming)

First half of the course

1) Understand q algorithms by comparison to deterministic and especially probabilistic algorithms

2) Understand the role of interference in quantum speedups

3) Learn to program quantum computers

Second half of the course

1) Implementing modern quantum algorithms
2) Understanding where we think the "killer applications" are for q. algorithms
3) Understanding the limitations of efficient q. computation

Lecture outline

In this lecture I want to show you why utilizing quantum phenomena might be useful for computation. To do this we'll introduce and use a toy model of computation known as interference computation. This will illustrate the core features of q. computation and q. algorithms

To do this, we'll need to cover the following subjects

- What is q. computation? (briefly)
- A weird dice example illustrating interference
- Algorithm types and efficiency
- Deterministic vs Probabilistic vs Interference computation
- The black box model
- Solving a problem with interference

This is what we'll do in this lecture and we'll keep working with the interferena model in the next 2 lectures as well.

What is quantum computation?

- Computation that leverages the rules of quantum mechanics.

- Information is not represented as bits but as qubits; elementary operations of a q. computer (gates) manipulate/transform these qubits; qubits can be turned into bits through a probabilistic process called measurement

- Idea of q.c. by Feynman and others in the 1980s
- Simulate a q. physical system using another q. physical system (this simulator is the q. computer)
- The field "took off" after the discovery of Shor's algorithm in 1994

— Shor's algorithm is a quantum algorithm for factoring integers (given $N = p \cdot q$, with p and q primes, find p and q). This problem seems very difficult classically (no known efficient classical algorithm). This is good because a lot of crypto used on the Internet relies on factoring being a hard problem. But Shor's algorithm is an efficient q alg for factoring.

If we had a large enough q. computer we could use it to "break" most of the crypto protocols used online.

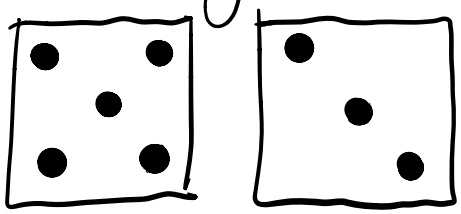
— Other interesting and efficient q. algorithms have also been discovered and we'll be covering some of them

Before we delve into q. computation, we're going to spend some time understanding a different model of computation called interference computation.

To get a feel for interference, let's look at an example. You shouldn't take this example too seriously, but you should also keep it in mind throughout the course :)

Weird dice

Consider an ordinary pair of dice



i j

We'll label the value of each die as i and j, respectively

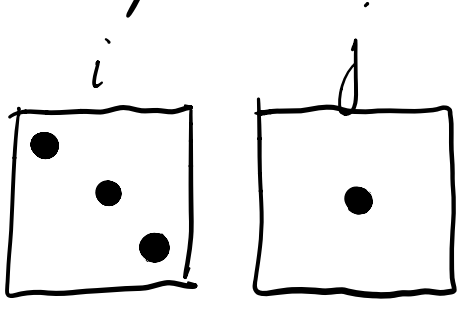
$1 \leq i, j \leq 6$

We'll assume the dice are fair, so each value has probability 1/6 (also die rolls are independent)

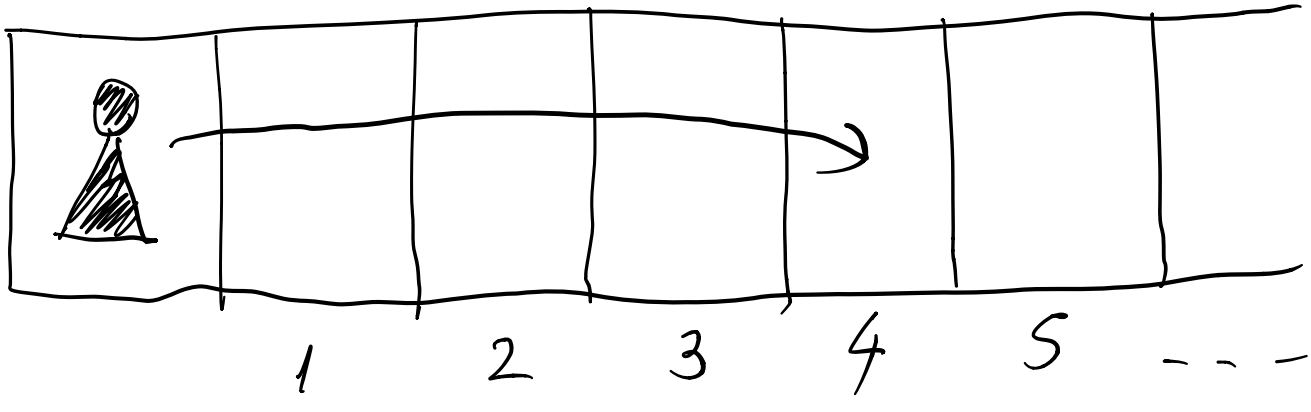
What is the probability of rolling (i, j) for the pair?

$$Pr(i, j) = \frac{1}{6} \cdot \frac{1}{6} = \frac{1}{36}$$

In games like Monopoly we usually care about the sum of the values, $i+j$, since that gives us the distance we move our game piece



$i+j=4$



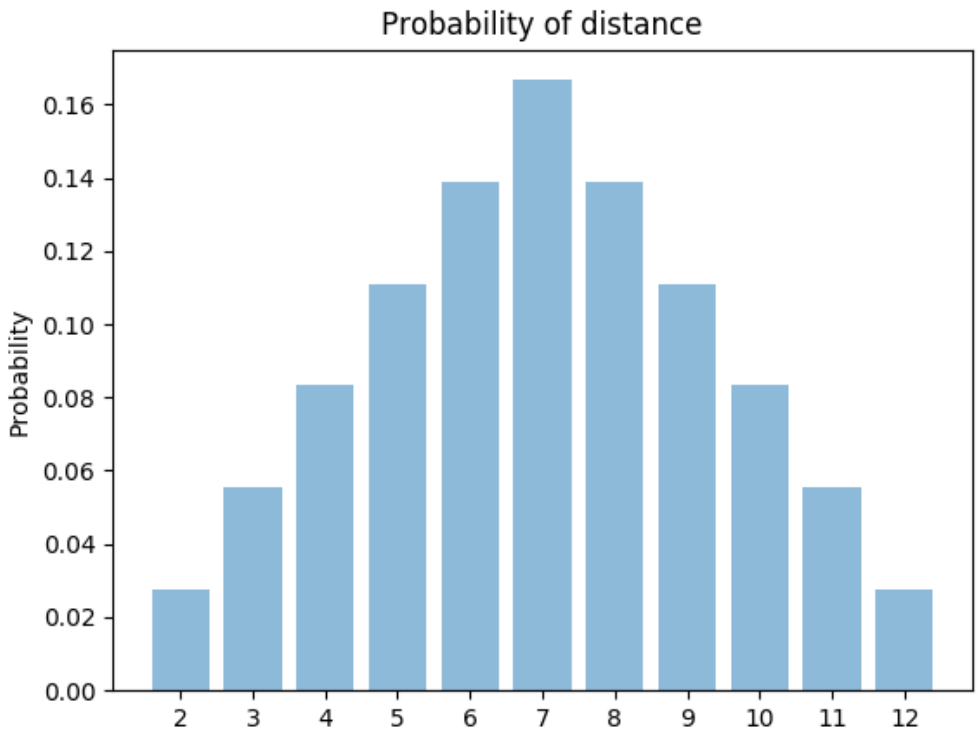
What is the probability of getting distance d?

Note that $2 \le d \le 12$

$$\Pr(d) = \sum_{\substack{i,j \\ i+j=d}} \Pr(i,j)$$

Eg: $\Pr(d=6) = \Pr(1,5) + \Pr(2,4) + \Pr(3,3) +$
$+ \Pr(5,1) + \Pr(4,2) = \frac{5}{36}$

If we were to plot probs (see code)



We're now going to do something apparently crazy.

Suppose we have a pair of weird dice for which it is the case that $\Pr(i,j) = -\Pr(j,i)$, when $i \neq j$.

What does this even mean? We won't worry about the interpretation of negative probabilities for now. Let's see what happens if we compute $\Pr(d)$ using the same formula from before

$$\Pr(d) = \sum_{\substack{i,j \\ i+j=d}} \Pr(i,j)$$

But now note what happens if d is odd.

E.g. $\Pr(d=5) = \Pr(1,4) + \Pr(2,3) + \Pr(4,1) + \Pr(3,2)$

$\Pr(4,1) = -\Pr(1,4)$ $\quad$ $\Pr(3,2) = -\Pr(2,3)$

So $\Pr(d=5) = 0$!

In fact for all odd d $\Pr(d) = 0$

What about even d?

Eg $\Pr(d=4) = \Pr(1,3) + \Pr(2,2) + \Pr(3,1)$

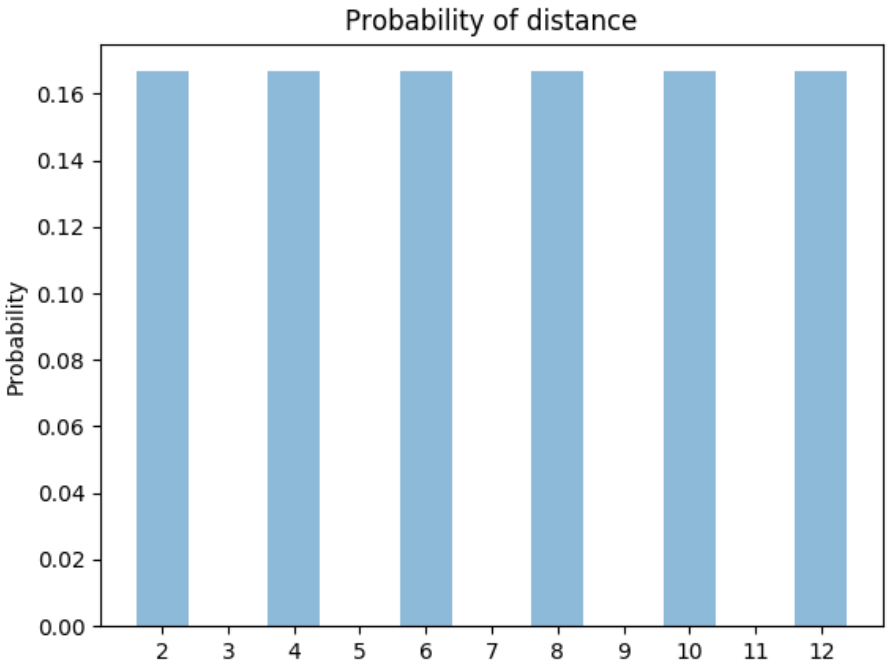
$= \Pr(2,2)$

In general, for even d, $\Pr(d) = \Pr\left(\frac{d}{2}, \frac{d}{2}\right)$

We'll assume $\Pr(i,i) = \Pr(j,j)$ so that everything normalizes nicely. Since

$$\sum_{d=2}^{12} \Pr(d) = 1$$

we'll have that $\Pr(i,i) = \frac{1}{6}$ and so



This is the idea of interference! It's as if we can only observe the sum of the dice but not their individual values. Combinations cancel out with their "mirror reflections".

Here are 2 important conclusions from this

(1) Allowing for dice roll combinations to cancel out other combinations made certain distances not occur at all (destructive interference) whereas other distances have higher probability of occurring than before (constructive interference)

(2) If you were given the prob distribution from above and were told it came from dice rolls, what would you conclude? The dice are perfectly correlated! Each die still has prob 1/6 for each value $1 \leq i \leq 6$

but the dice always appear as (i, i)

This correlation resulting from interference is similar to the quantum effect of entanglement.

Observation 1) is the one that's relevant in the context of algorithms. It basically tells us that if our goal is to, say, roll a 2, we should use weird dice since the expected number of rolls is 6 whereas for ordinary dice it's 36!

As I mentioned, though, we shouldn't take this too seriously. Negative probabilities don't make sense! However, we will mimic their effect with what are called amplitudes. More on that later!

Algorithm types and efficiency

What is an algorithm?

A sequence of instructions that solves a problem (or performs a computation).

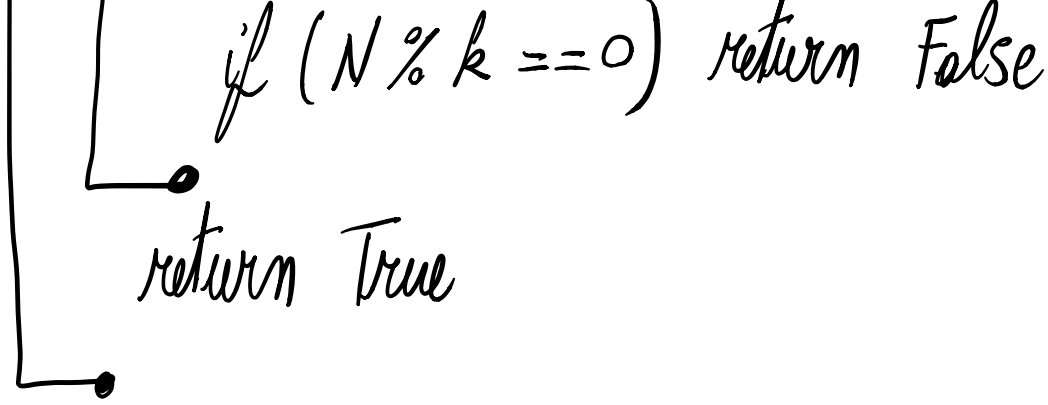
E.g. Primality testing

In: N, positive integer

Out: True, if N is prime
False, otherwise

Most naive algorithm ever for primality testing

```
def isPrime(N)
    if(N<2) return False
    for k in range(2, N)
```



What is the runtime of this algorithm? I.e. how many elementary instructions will it execute in the worst case?

We can see that if N is actually prime, then k goes all the way up to N-1.

This means that in the worst case the runtime is $\Theta(N)$

Letting n be the number of bits of N (so $n = \lceil \log N \rceil$) we have $2^{\Theta(n)}$ runtime

This isn't great! Runtime that is exponential in the size of the input is viewed as inefficient computation.

What is efficient computation?

Throughout the course we will consider efficient

algorithms to be those that run in time that is polynomial in the input size, in the worst case!

E.g. runtimes like n, $n/2$, $10n^2$, n^3, $n^5/100$ etc

There are of course problems with this definition (is n^{100000} efficient?), but for the algorithms we see in practice it works pretty well.

So is there a polynomial-time algorithm for primality testing?

Yes! There is a famous probabilistic algorithm* called the Miller-Rabin algorithm.

We won't be looking at it, but we'll use this to segue into a discussion about deterministic vs probabilistic algorithms

*There's also a deterministic one, but it's less efficient.

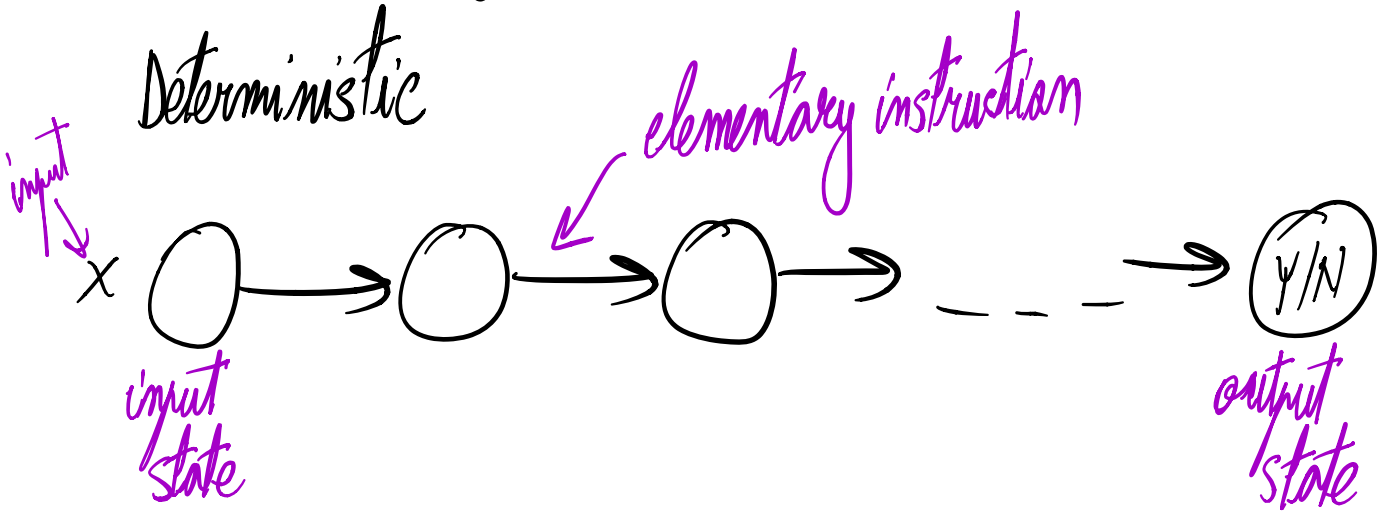
Deterministic vs probabilistic algorithms

- Deterministic algorithm - always produces the correct output to the problem it's solving
- Probabilistic algorithm - uses randomness and produces correct output with high probability, but can also err

Conceptually they look like this

(we're going to assume we're solving a decision problem i.e. a problem for which the answer is Yes or No, like primality testing)

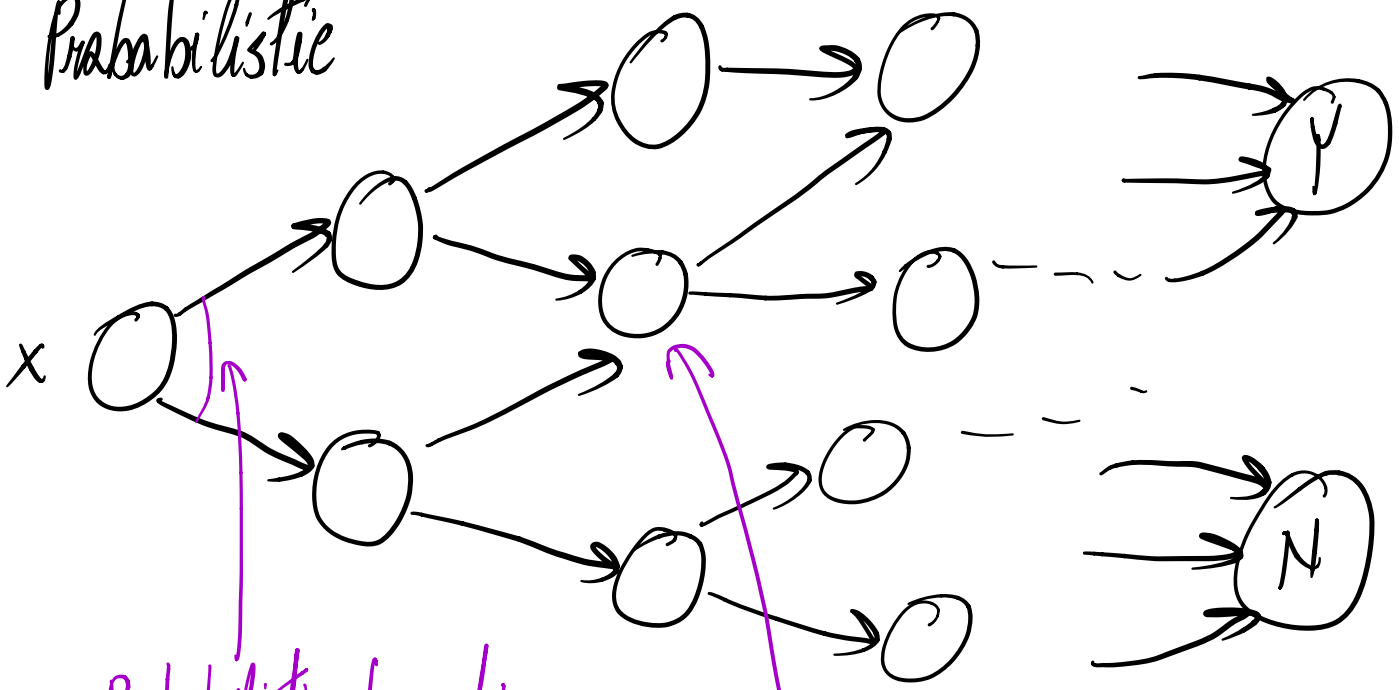
Deterministic



Each ○ represents a memory state of the computer

Each → represents an elementary instruction
(we'll also call these transitions)

Probabilistic



Probabilistic branching
(assume each branch has prob 1/2)

Can arrive at this state from 2 different paths.

In the probabilistic case, it's as if the computer flips a coin every now and then and based on the outcome decides to execute one instruction or another.

An important point is that you can get to certain states

from different paths. Notably there are 2 possible output states (Y) for a Yes outcome and (N) for a No outcome.

All paths leading to Yes terminate at (Y)

All paths leading to No terminate at (N)

For the probabilistic alg. to be useful it must be that we get correct result with high probability. What does high mean? $\frac{1}{2}$ + const Why? We can amplify to as close to prob 1 as we like by repeating the alg. many times and taking a majority vote.

So...

For any input, x, a probabilistic algorithm produces the correct output with probability at least $\frac{1}{2} + \epsilon$.

Note 1: $\frac{1}{2} + \frac{1}{poly(n)}$ is the more general form

Note 2: We'll usually take that probability to be $\frac{2}{3}$

This means that when the answer is Yes, $Pr(\text{Yes}) \geq \frac{1}{2} + \epsilon$ and when the answer is No, $Pr(\text{No}) \geq 1/2 + \epsilon$

This is called bounded error.

But what are $Pr(\text{Yes})$, $Pr(\text{No})$?

$$Pr(\text{Yes}) = \sum_{P \in \text{Paths}(x, Y)} Pr(P)$$

$$Pr(\text{No}) = \sum_{P \in \text{Paths}(x, N)} Pr(P)$$

Where $\text{Paths}(x, Y)$ is the set of all paths that start at input x and end in the Yes output state.

$\text{Paths}(x, N)$ is the set of paths starting in the input state and ending at the No output.

What is the probability of a path P? It's just the product of probabilities of all transitions along the path.

If we imagined that all paths had equal probability then what we want is for most paths to terminate in the node corresponding to the right answer.

At this point, it's useful to think back to the dice example. The probability of a distance d is like the probability of our Yes and No nodes.

This will be our segue into interference computation where we want paths to cancel each other out (like the weird dice outcomes)

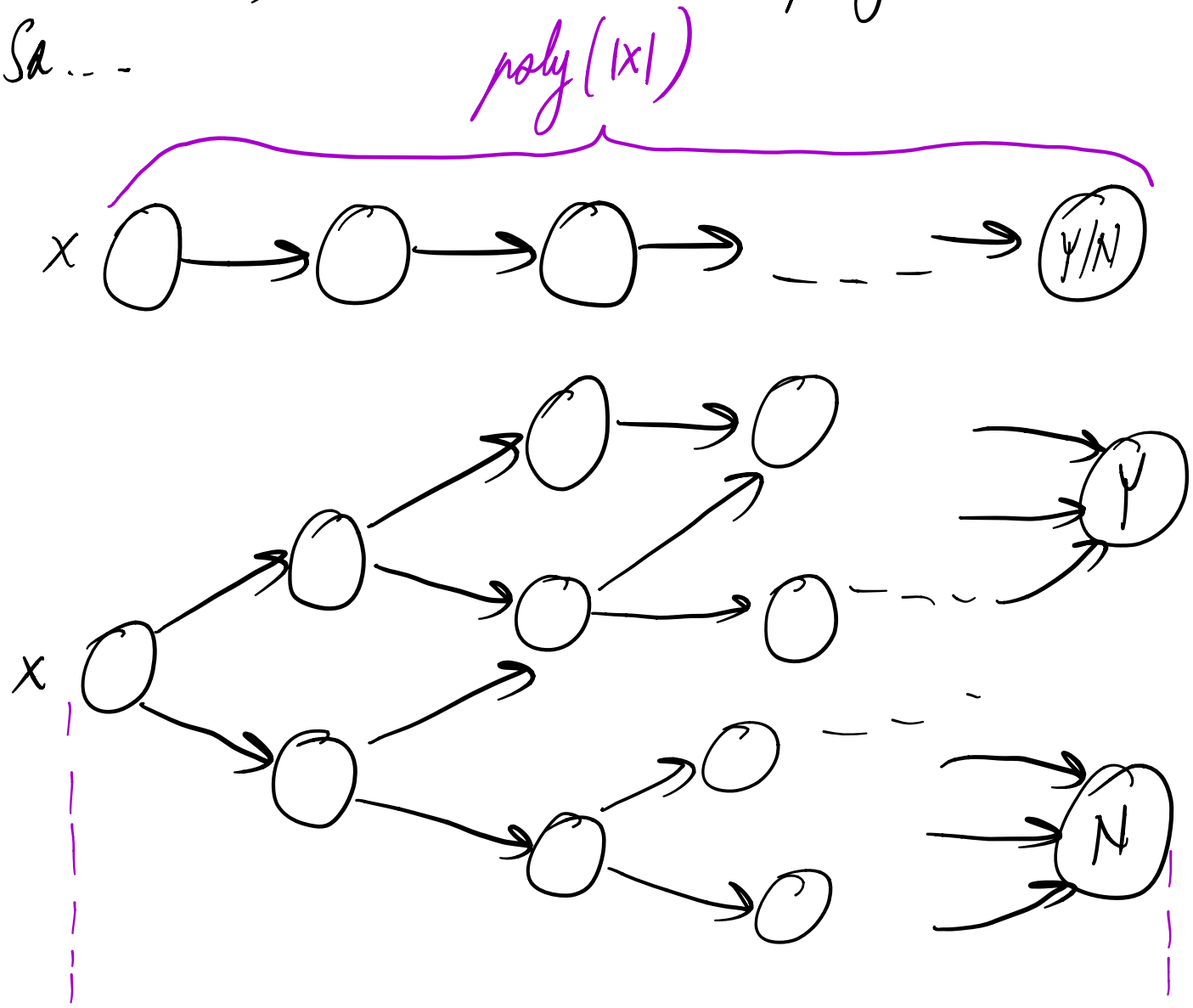
But first we need to talk about efficiency.

What about efficiency?

The runtime of the algorithm is the length of the path from the input state to the output state(s).

Since we are interested in efficient algorithms, we want this length to be some polynomial in the length of x (denoted $|x|$) which we write as $poly(|x|)$

So...



poly(|x|)

We refer to these as deterministic polynomial time algorithms (DPT) and bounded-error probabilistic polynomial time algorithms. (PPT), respectively.

The set of problems we can solve with the former is denoted P and for the latter it's BPP.

More on these in a later lecture, but for now note that $P \subseteq BPP$. Why? If in the PPT algorithm we set all probabilities to either 1 or 0, we get a deterministic algorithm. So PPTs are more general than DPTs.*

* In fact it's believed that P=BPP for complicated reasons that we won't get into, but just imagine using pseudo-randomness instead of true randomness

Interference computation

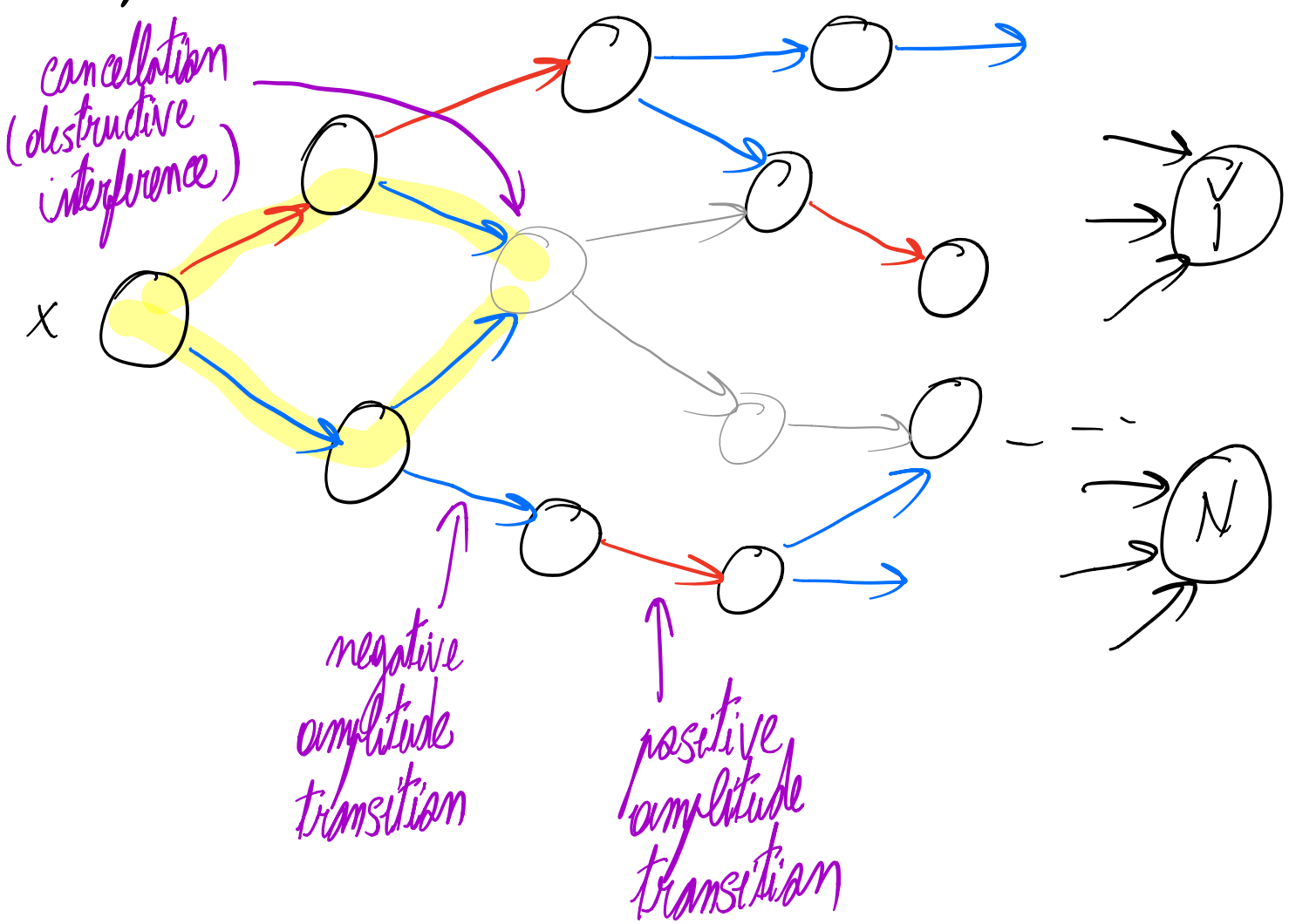
In analogy to the weird dice example, we're going to consider interference computation to be like probabilistic computation but where to each path we associate a number called an amplitude which can be either positive or negative (and in general is a real number).

The amplitude of a path is the product of the amplitudes of all transitions along that path.

The amplitude of a node is the sum of the amplitudes of all paths leading to that node (and starting at the input node).

The probability of a node is the absolute value of the amplitude of the node, suitably normalized.

Before writting this down more formally let's draw a picture



Note the 2 paths that meet in the middle gray node. One has a pos. amplitude transition and a neg. amplitude one and so the amplitude of that path is negative. The other has 2 neg amplitude transitions and so has positive amplitude. If these are equal in absolute value they will cancel and so there will be no contribution to the paths from the gray node!

Takeaways

$$Amp(Yes) = \frac{1}{N} \sum_{P \in Paths(x,Y)} Amp(P)$$

$$Amp(No) = \frac{1}{N} \sum_{P \in Paths(x,N)} Amp(P)$$

for some suitable N that ensures that

$$Amp(P) = \widetilde{\prod_{t \in Trans(P)}} Amp(t), \text{ where } Trans(P)$$

is the set of all transitions on the path P

$$Pr(Yes) = |Amp(Yes)|, \quad Pr(No) = |Amp(No)|$$

$$Pr(Yes) + Pr(No) = 1$$

As before we require that $\forall x$, if answer to the problem is Yes, $Pr(Yes) \geq \frac{1}{2} + const$ and if the

answer is N_2, $\Pr(N_2) \geq \frac{1}{2} + \text{const}$.

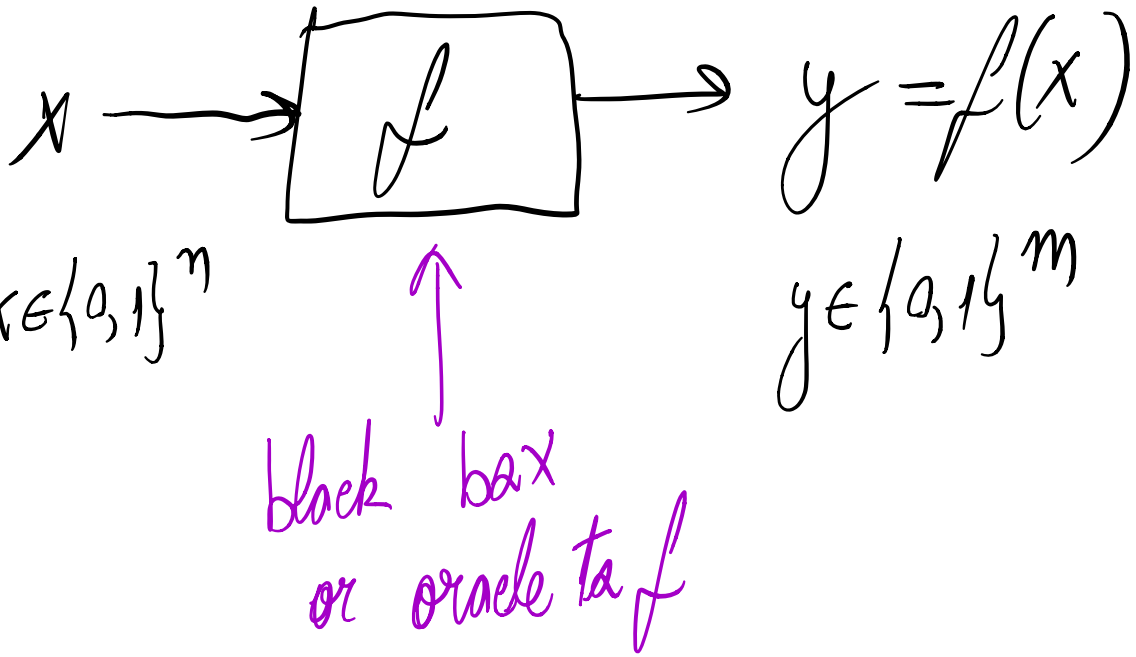
DISCLAIMER: This is not quite how q. computing works. It will help us understand q. computing, but the interference we're considering here is more general.

At this point things might still be a bit vague, so we're going to make them more precise so that we can actually write code to perform interference computation. Of course, we don't have an "interference computer", so we'll have to simulate the interference process.

To do this, let's first discuss a useful model for studying algorithm efficiency.

The query / black box / oracle model

In this model we're given access to a function $f: \{0,1\}^n \to \{0,1\}^m$ and asked to determine a property of this function using as few queries to the function as possible. A query is simply an evaluation of the function. We think of the function as being given to us like a black box. We can only query it, but we don't have the "code" implementing it



The Deutsch-Jozsa problem

Input: black box access to $f: \{0,1\}^n \to \{0,1\}$

promised that f is either

- constant: $\forall x, y \in \{0,1\}^n \quad f(x) = f(y)$
- balanced: on half of all inputs in $\{0,1\}^n$ f is 0 and on the other half it's 1

(more formally let $S_0, S_1 \subseteq \{0,1\}^n$ s.t $S_0 \cap S_1 = \emptyset$,

$|S_0| = |S_1| = 2^{n-1}$ $\quad \forall x \in S_0 \quad f(x) = 0$

$\forall x \in S_1 \quad f(x) = 1$)

Output: Yes if f is constant

No if f is balanced

Deterministic case

How many queries do we need to solve this problem with a deterministic algorithm?

$$2^{n-1}+1$$

Why? Assume f is balanced and we make k queries $x_1, x_2 \dots x_k$. If $k \leq 2^{n-1}$ we could be unlucky and get $f(x_1) = f(x_2) = \dots = f(x_k)$.
So we need at least $2^{n-1}+1$ different queries to "break the tie".

Probabilistic case

Consider the following probabilistic algorithm.

$x_1, x_2 \leftarrow_U \{0,1\}^n$
$(x_1 \neq x_2)$

$y_1 = f(x_1)$, $y_2 = f(x_2)$

if $y_1 == y_2$ return Yes

else return No

This clearly works when f is constant. What about when f is balanced?

Fix the value of y_1. What is the probability that $y_2 \neq y_1$?

$$\frac{2^{n-1}}{2^n - 1} = \frac{1}{2 - \frac{1}{2^{n-1}}}$$; while this is $> \frac{1}{2}$,

it's not $\frac{1}{2} + \epsilon$, since as n increases this gets inverse exponentially close to $\frac{1}{2}$

So this isn't a good algorithm. How to fix it? Make 3 queries instead

$x_1, x_2, x_3 \leftarrow_U \{0,1\}^n$

$(x_1 \neq x_2, x_2 \neq x_3, x_1 \neq x_3)$

$y_1 = f(x_1), \; y_2 = f(x_2), \; y_3 = f(x_3)$

if $y_1 == y_2 == y_3$ return Yes

else return No

In this case, after fixing y_1, the probability that $y_1 = y_2 = y_3$ is $\sim \frac{1}{4}$. This is the probability of incorrectly declaring the balanced function to be constant.

So prob of deciding correctly is $> \frac{3}{4}$

Interference case

We still haven't defined what the interference model is, so let's do that now

interference computation = regular classical comp + the function interf1 defined as follows

interf1(fun : $D \to \{-1, +1\}$, Dom $\subseteq D$, $N > 0$)

Let $S = \frac{1}{N} \left| \sum_{x \in Dom} fun(x) \right|$

if ($S \geq 2/3$) return True

if ($S \leq 1/3$) return False

return random ([True, False])

Let's parse this definition. interf1 is a function that takes as arguments

- a function fun : $D \to \{-1, +1\}$

This should be thought of as a function that determines whether a path in the computation tree ends up

having positive or negative amplitude.
It should be efficiently computable (i.e. computable in polynomial time or with poly-many queries to the black box oracle)

- a domain/set Dom that is a subset of D
In general this set should be very large (again, think of this as the set of all paths in the computation tree); However each element in Dom should be computable in poly time (or with poly-many queries)

- a normalization constant N, whose purpose is to ensure that

$$0 \leq S = \frac{1}{N} \left| \sum_{x \in Dom} fun(x) \right| \leq 1$$

What interf1 is doing is essentially computing the interference

of outcomes of fun and telling you whether the interference is strongly constructive ($\geq 2/3$) or strongly destructive ($\leq 1/3$). In between we don't care what happens.

Note: The reason for the name interf_1 is because we'll encounter interf_2 in the next lecture :)

Here's the main point:

> We view one call of interf_1(fun, Dom, N) as a single call to fun

Of course, this is not true as we can see from the implementation of interf_1, but this is what we mean by simulating interference computation. In the interference model we have the same elementary instructions as in regular computation + interf_1 which we view as having the same runtime as 1 call to fun.

Let's see how we can solve DJ using interf1

Let $fun(x) = (-1)^{f(x)}$

(so $fun: \{0,1\}^n \to \{-1,+1\}$)

$Dom = \{0,1\}^n$

$N = \frac{1}{2^n}$

return interf1 (fun, Dom, N)

Case 1: f is constant; then $fun(x) = 1$ or -1 for all values of x

$$\Rightarrow S = \frac{1}{2^n} \left| \sum_x fun(x) \right| = \frac{2^n}{2^n} = 1$$

$S \geq \frac{2}{3}$ so we return True

Case 2: f is balanced; then $fun(x) = 1$ for half the

values of $x \in Dom$ and $f(x) = -1$ for the other half

$$\Rightarrow S = \frac{1}{2^n} \left| \sum_x f_{un}(x) \right| = 0$$

$S \leq \frac{1}{3}$ so we return False

Success! Our interference comp. is correctly deciding the DJ problem with only one call (in our model) to f!

This is the magic of interference.

Next time we'll look at other problems where we get greater speedups compared to the probabilistic model (here the speedup is modest: 1 query vs 3)