Quantum Computation **(CMU 18-859BB, Fall 2015)**

Lecture 23: Introduction to Quantum Complexity Theory

November 31, 2015

Lecturer: Ryan O'Donnell *Scribe: Will Griffin*

1 REVIEW: CLASSICAL COMPLEXITY THEORY

Usually, when doing complexity theory, we look at decision problems, where the output is 0 or 1. This is because it makes the problems simpler to analyze, and there is not much loss of generality since solutions to decision problems can be efficiently turned into solutions for function problems, where the output is more complex. Using this, we can define a general function to be analyzed:
$f : \{0,1\}^* \rightarrow$, where the $*$ implies it's domain can be any set of binary strings, and 1 corresponds to yes, 0 to no.
This can be reformulated in terms of formal languages, where for some decision problem $f(x)$, the language L corresponding to f is the set of all binary strings x such that $f(x) = 1$, so: $L = \{x : f(x) = 1\}$.
Examples:

$CONN = \{x : x$ encodes a connected graph$\}$

$PRIMES = \{x : x$ encodes a prime number$\}$

$SAT = \{x : x$ encodes a boolean formula that is satisfiable$\}$

$FACTORING = \{(x, y) : x$ has a prime factor between 2 and $y\}$

Of these examples, $CONN, PRIMES$, and SAT are naturally decision problems. For $FACTORING$, however, it seems more natural to return a prime factor of x. However, this is not an issue, since prime factors of x can be found by changing y and using it to binary search the numbers between 2 and x, which takes linear time in the number of bits of x.

2 COMPLEXITY CLASSES

A Complexity Class is a set of Languages with related complexity. First we will define 4 of them, and show how they are related.

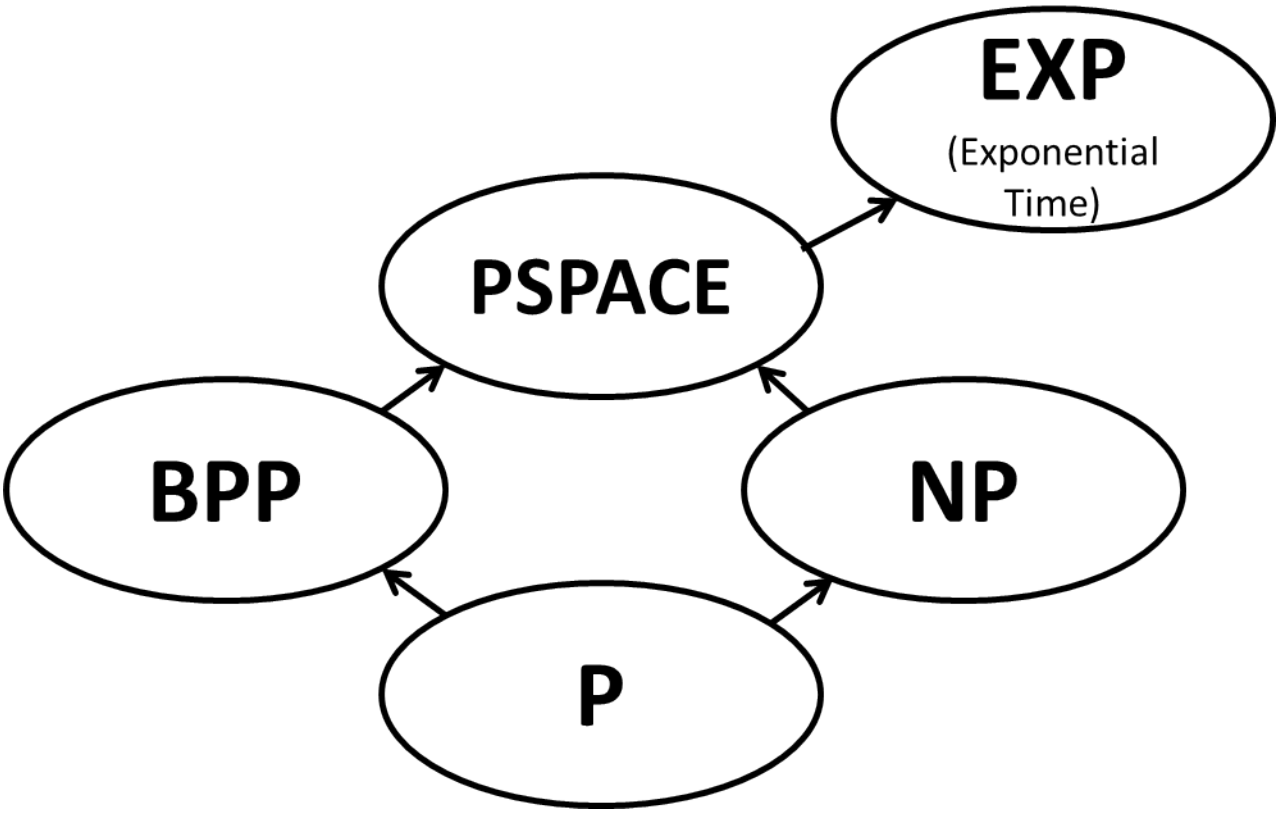
P = { Languages L : the problem $x \in L$ is decidable in $Poly(n)$ time by a deterministic algorithm }.
In this definition, a deterministic algorithm is a standard Turing machine.
P is considered efficient. $CONN$ (shown in class) and $PRIMES$ are in P [AKS04]. The others are not known to be in P, though most complexity theorists think they are not.

$PSPACE$ = { Languages L : the problem $x \in L$ is decidable in $Poly(n)$ space deterministically}.
Space can be defined in terms of memory cells or tape spaces in a Turing machine.
All of the examples are in $PSPACE$, as they can all be computed with brute force methods using $Poly(n)$ space.
P is in $PSPACE$ since an algorithm can only use $Poly(n)$ space in $Poly(n)$ time.

BPP (Bounded Error, Probabilistic, Polynomial) = $\{L$: the problem $x \in L$ is decidable in $Poly(n)$ time by a randomized algorithm$\}$
Here, randomized algorithm means a standard Turing machine that has access to a 'coin flipper', which can output 0 or 1 each with probability 1/2.
Decided, for this class, is usually taken to mean that the probability of accepting x is greater than 3/4 is $x \in L$, and is less than 1/4 if x is not in L, so we say an algorithm in BPP decides f if $\forall x, Pr[Alg(x) = f(x) > 3/4$.
The 3/4 is fairly arbitrary, and it can be changed to any θ with the restriction that: $\theta \leq 1 - 2^{-n}$ and $\theta \geq 1/2 + 1/Poly(n)$ without changing the class at all. This can be shown using Chernoff bounds, (it was a problem on homework 1) and is called error reduction, and can be accomplished by running the algorithm multiple times and taking the majority value of the guesses. However, if $\theta < 1/2 + 1/Poly(n)$ it can take exponentially many runs to get to 3/4 probability, and if $\theta > 1 - 2^{-n}$, it can take exponentially many runs to reach that probability. Forcing those constraints on the output is what is meant by 'bounded error'.
BPP is, informally, that class of problems with efficient random algorithms.
$CONN$ and $PRIMES$ are in BPP, is not known whether the other examples are, though most people think they are not.
$P \subseteq BPP$ because a BPP algorithm can just not use its randomness. It is believed that $P = BPP$, but unproven. This is because of excellent pseudo-random algorithms that can effectively simulate truly random computation on a deterministic machine.
$BPP \subseteq PSPACE$. Since any randomized output takes $Poly(n)$ time and space to run, a $PSPACE$ algorithm can try all of them, and exactly compute the probability that the BPP algorithm would get the right answer.

NP (Non-deterministic Polynomial Time) = $\{L : L$ has a one-way, deterministic, $Poly(n)$ time proof system$\}$
The full definition of NP relies on a $Prover$ and a $Verifier$. The $Prover$ is trying to convince the $Verifier$ that some given binary string x of length $Poly(n)$ and language L, that $x \in L$.

The $Verifier$ wants to accept the proof if and only if $x \in L$.
Formally, for some x, L, the $Prover$ sends some string $\pi \in \{0,1\}^{Poly(n)}$ to the $Verifier$. The $Verifier$ computes on (x, π) in deterministic polynomial time and answers $\{0, 1\}$ (1 for accept), with the rules:
$\forall x \in L$, $\exists \pi$ such that $Verifier(x, \pi) = 1$
$\forall x \notin L$, $\forall \pi$ $Verifier(x, \pi) = 0$
When $x \in L$, the $Verifier$ and $Prover$ are essentially working together, trying to find some string that shows $x \in L$.
When $x \notin L$, the $Prover$ is essentially working as an adversary, trying to find some string that the $Prover$ will accept even though the $Prover$ shouldn't.
$P \subseteq NP$, since the $Verifier$ can just ignore the proof and compute the question in $Poly(n)$ time.
It is not known if $BPP \subseteq NP$ (but it clearly is if $BPP = P$).
$NP \subseteq PSPACE$. Since each possible proof π can be checked in $Poly(n)$ time and space, a $PSPACE$ algorithm can check them all and accept if any of them accept.
All of the examples are in NP.
$CoNP$ may not be in NP. For example, $UNSAT = \{x : x$ encodes a boolean circuit with no satisfying assignment$\}$ may not be in NP.
The relationship between the different classes shown is in the diagram, which also shows that they are all contained in EXP, the class of problems that take exponential time to solve.



3 QUANTUM COMPLEXITY

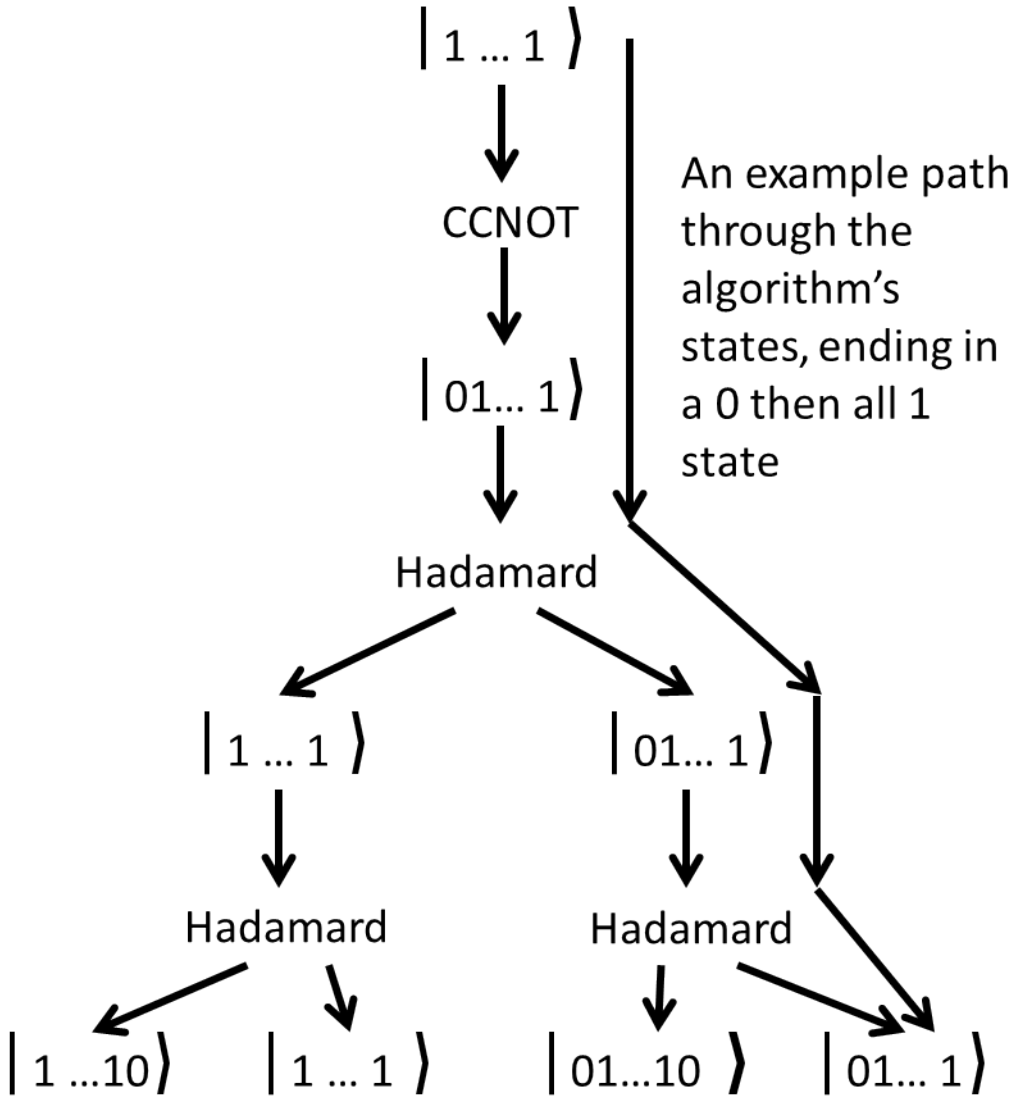
Since we've gone over Classical Complexity, where does Quantum Complexity fit in?
First, our model for quantum algorithms is quantum circuits, not Turing machines. Quantum Turing machines can be defined, but they are confusing and difficult to work with. Each circuit, however, can only take in inputs of one size, so it is not obvious how to define complexity for quantum circuits in a way that allows us to compare them to Turing machines that can take arbitray input sizes.
To overcome this issue, we use a model where a BPP algorithm writes down a quantum circuit with hard coded inputs, and the measurement at the end gives the result.
Then define:
BQP (bounded-error, quantum, polynomial time) = $\{L : \exists$ a BPP algorithm that can write down a quantum circuit with hard-coded inputs that can decide $x \in L\}$ [BV97].
In this definition, decide means that when the q-bits are measured at the end of the quantum circuit, the chance of seeing the wrong answer is less than $1/4$ for any x. This is essentially the same as deciding $x \in L$ for BPP, except that a measurement needs to be done.
Defining circuit complexity as the complexity of a Turing machine that can write down the circuit is robust, and used in classical complexity as well. For example, $P = \{L : \exists$ a P algorithm that can write down a classical circuit that can decide $x \in L\}$ and
$BPP = \{L : \exists$ a P algorithm that can write down a classical randomized circuit that can decide $x \in L\}$.
In addition, the definition is robust to changes in the set of quantum gates used for computation, so using a different set of quantum gates does not change the definition of the class [DN05].
$CONN, PRIMES$, and $FACTORING$ are in BQP. BQP contains P and BPP and $BQP \subseteq PSPACE$. The relationship between BQP and NP is unknown, though it is widely believed that there are problems in NP not in BQP and problems in BQP not in NP.
The evidence for those beliefs is based on oracles - there are oracles relative to which BQP has problems not in NP and vice versa.
Additionally, it is believed that BQP is not a part of the Polynomial Time Hierarchy. [Aar10]

Theorem 3.1. $BQP \subseteq PSPACE$

Proof. Suppose a language $L \in BQP$, so there is a $Poly(n)$ time algorithm that, on input x, writes down a Quantum circuit C_x with inputs hardcoded to $|1\rangle$, and when the 1^{st} bit of the output of C_x is measured, it gives $f(x)$ with probability greater than or equal to $1/4$.
C_x starts in the state $y_1, y_2, \ldots y_m = |111\ldots11\rangle$, $m = Poly(n)$.
Then C_x applies a series of $CCNOT$ and $Hadamard$ (H) gates to the bits, and then the 1^{st}

bit is measured.
As drawn below, the state can be explicitly tracked, and a classical machine can track the full state through the operation of each gate. $CCNOT$ gates are easy to track because they are deterministic, but H gates force the classical computer to split each partial state it is tracking into two states, doubling what it must keep track of.
Then, if there are h Hadamard gate, at the end of the computation there will be 2^h branches, each corresponding to a different path through the circuit (even though some of the paths may give the same measurement outcome). Then let $|p\rangle$ be a final state. p can be seen as a path through the 'state tree' drawn, and at each branching off point, there is a certain probability amplitude assigned to p at that point (in this case it's always $1/\sqrt{2}$ since we're using H gates).

Example 'State Tree' for an extremely simple circuit:



Then, at the end of the computation, the final state can be written as a sum over all the paths times their amplitudes, so $|final\rangle = \sum_p amp(p)\,|p\rangle$.
$amp(p) = \left(\frac{1}{\sqrt{2}}\right)^h sign(p)$, where $sign(p)$ is the product of the signs of the amplitude of p. Then $|final\rangle = \left(\frac{1}{\sqrt{2}}\right)^h \sum_p sign(p)\,|p\rangle$. Then the final amplitude of a state $|s\rangle = \left(\frac{1}{\sqrt{2}}\right)^h \sum_{p:|p\rangle=|s\rangle} sign(p)$. Then the probability of measuring the state $|s\rangle$ is the squared magnitude of the amplitude, so $Pr(s) = 2^{-h}\sum_{p,p'} sign(p)sign(p')$, where the sum over p, p' is over $|p\rangle = |s\rangle\,, |p'\rangle = |s\rangle$.
What we want is the probability that we measure a $|0\rangle$ or $|1\rangle$ for the first bit. But since we know the probability of measuring a state $|s\rangle$, we can just sum over all states $|s\rangle$ with a $|1\rangle$ in the first bit. Then $Pr(1) = \sum_{p,p'} sign(p)sign(p')$, where the sum is over paths p, p' such that $|p\rangle = |p'\rangle$ and the first bit is $|1\rangle$.
Additionally, we can use the formula for $Pr(C_x accept)$ to then get: $Pr(C_x accept) - Pr(C_x reject) = 2^{-h}\sum_{p,p'} sign(p)sign(p')(-1)^{1-state(p)}$, where $state(p)$ is the first bit of $|p\rangle$, and the sum is over $|p\rangle = |p'\rangle$. If $x \in L$, this is greater than $1/2$, and if $x \notin L$, this is less than $-1/2$.
This shows how to calculate the output probability classically, but since there are exponentially many paths, it would take an exponential amount of space to store the path information. To make an algorithm to do this in $PSPACE$, the key is to calculate the contribution to the probability from each path, which can be done in $Poly(n)$ time and space since there are $Poly(n)$ gates, and add them up, which allows us to find $Pr(1)$ with a $PSPACE$ algorithm taking exponential time.

□

4 BQP and PP

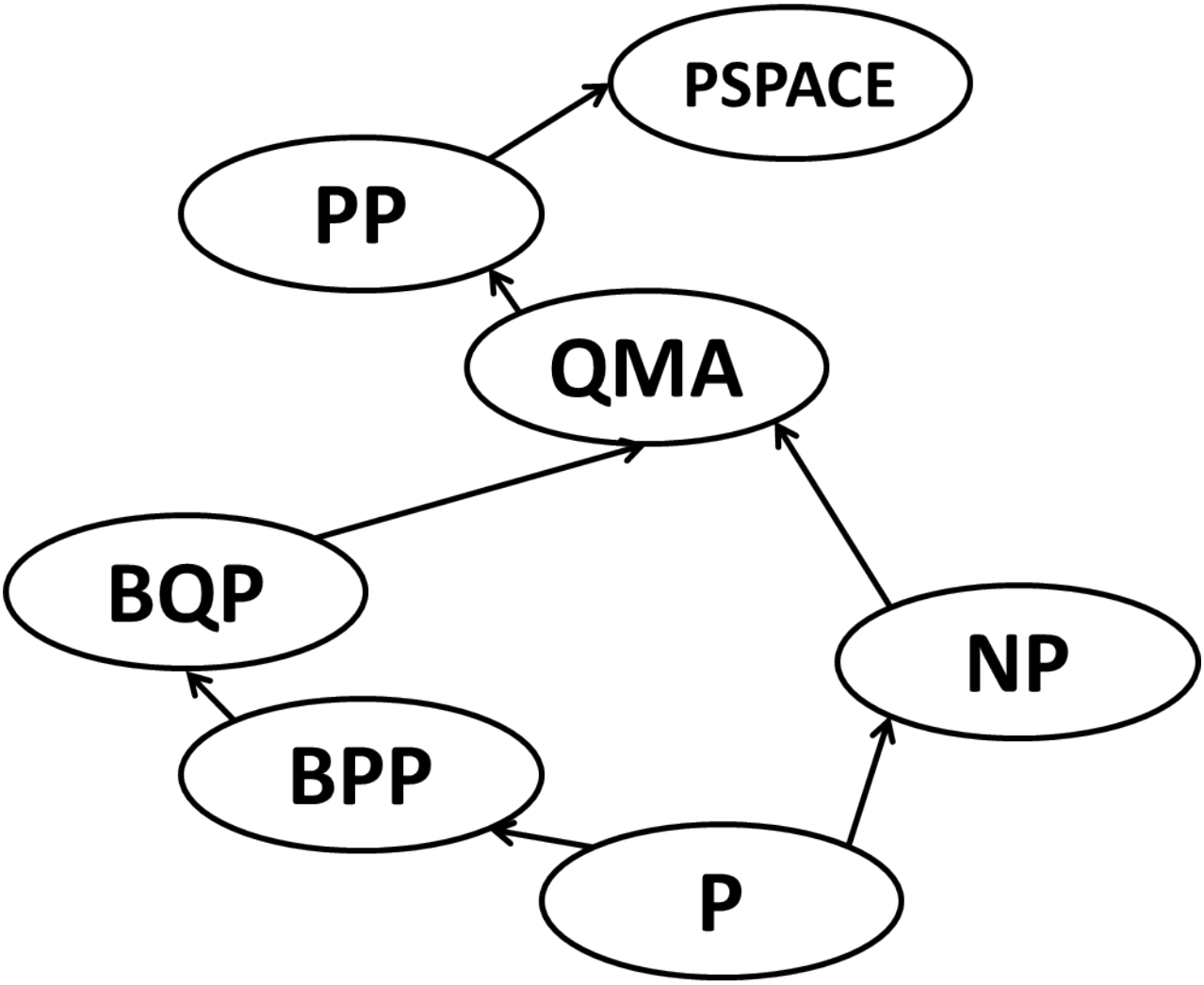
PP (Probabilistic Polynomial) = $\{L : x \in L$ is decidable by a probabilistic polynomial time algorithm$\}$.
Here, decidable means that the probability of the algorithm being wrong is less than $1/2$. This is different than BPP and BQP, which force the error to be less than $1/2$ by at least a $1/Poly(n)$ amount.
Because of that difference, PP lacks efficient error amplification, so algorithms in PP are not necessarily efficient, since it can take exponentially many runs to get a high success probability. This makes PP less useful than BPP as a complexity class.
$PP \subseteq PSPACE$, this can be seen in the same way as $BPP \subseteq PSPACE$, since a $PSPACE$ algorithm can just iterate through every possible random outcome and find the probability.
PP contains both NP and BQP. It also contains the quantum version of NP, QMA, which is analogous to NP except the $Prover$ sends a quantum states and the $Verifier$ decides to accept with a quantum algorithm.

Theorem 4.1. *$BQP \subseteq PP$ [ADH97]*

Proof. Given some quantum algorithm Q, the PP algorithm Alg first randomly computes the end state of two paths in Q, p and p' independently (by computing the quantum state of the system and randomly picking one choice whenever a Hadamard gate is used).
Then let $|p\rangle, |p'\rangle$ be the end states of the paths, and q be the probability that $|p\rangle \neq |p'\rangle$. If $|p\rangle \neq |p'\rangle$, Alg outputs 1 with probability $P > 1/2$, -1 with probability $P < 1/2$.
If $|p\rangle = |p'\rangle$, it outputs $sign(p)sign(p')(-1)^{1-state(p)}$, where $state(p)$ is the value of the first bit of the final state of path p. Then the expectation value of Alg is:
$E(Alg) = (1-q)2^{-h}\sum_{p,p'} sign(p)sign(p')(-1)^{1-state(p)} + q(2P-1)$,
and $2^{-h}\sum_{p,p'} sign(p)sign(p')(-1)^{1-state(p)}$ is $Pr(accept) - Pr(reject)$ as shown before.
If $x \in L$, then $E(Alg) > 0$.
If $x \notin L$, then $E(Alg) < 0$.
Then this shows that Q can be simulated by a PP algorithm by accepting with probability $P = 1/2 + Alg(x)$.

□

With all the complexity classes we have now defined, the picture of the complexity hierarchy is now:



5 A few more details about PP

Just wanted to add a few things about the complexity class PP, a rather strange class.

Super-minor, boring technical detail.

First, a super-minor and boring technical detail. For PP, we say that the machine "accepts" a string x if it outputs 1 with probability $> 1/2$. We say that it "rejects" a string x if it outputs 1 with probability $< 1/2$. What if, for some x, it outputs 1 with probability exactly $1/2$? Does that count as "accept" or "reject"? The answer is: Doesn't really matter; any reasonable convention you decide on gives the same class PP.
To see this, let's say we take the convention that outputting 1 with probability $1/2$ counts as "reject". Now suppose we have a randomized algorithm A running in time n^c that "decides"

language L in the sense that $x \in L \Rightarrow \mathbf{Pr}[A(x) = 1] > 1/2$ and $x \notin L \Rightarrow \mathbf{Pr}[A(x) = 1] \leq 1/2$. The first thing to note is that since A runs in time n^c, it flips at most n^c coins, and thus every possible outcome it has occurs with probability that is an integer multiple of 2^{-n^c}. In particular, when $x \in L$ we know that not only is $\mathbf{Pr}[A(x) = 1] > 1/2$, in fact it is $\geq 1/2 + 2^{-n^c}$. Now consider the following algorithm B. On any input x, it first flips n^{c+1} coins. If they all come up 1, then B immediately outputs 0. Otherwise, B simulates A. Note that B is still polynomial time. Furthermore, the effect of that first stage of flipping n^{c+1} coins is to slightly down-shift all of A's probabilities by roughly $2^{-n^{c+1}}$. So now we will indeed have that $x \in L \Rightarrow \mathbf{Pr}[B(x) = 1] > 1/2$ and $x \notin L \Rightarrow \mathbf{Pr}[B(x) = 0] < 1/2$. So you see, with this kind of trick, the exact tie-breaking convention in PP does not matter.

NP is in PP.

Now let me explain why NP is in PP. It's very simple. By the theory of NP-completeness, it is sufficient to show that SAT is in PP. Here is the algorithm. On input F, an m-variable Boolean formula (where $m < |F| = n$)), the PP algorithm guesses a uniformly random assignment $\alpha \in \{0,1\}^m$ and checks if it satisfies F. If so, the algorithm outputs 1. If not, the algorithm outputs a random coin flip. Now it's clear that if $F \in$ SAT then $\mathbf{Pr}$[output 1] $\geq 1/2 + 2^{-m}$ and if $F \notin$ SAT then $\mathbf{Pr}$[output 1] $\leq 1/2$. (In fact, it's $= 1/2$ in the second case.) This shows SAT is in PP, by the above super-minor boring discussion.

PP is closed under complement.

As you can see from the original $> 1/2$ vs. $< 1/2$ definition of PP, it is symmetrically defined with respect to yes-instances and no-instances; i.e., it is "closed under complementation"; i.e., if L is in PP then so is L^c. Hence UNSAT is in PP, which means that PP contains not just NP but also coNP (if you remember that class).

A "complete" problem for PP.

An easy-to-show fact in complexity theory: Let MAJ-SAT be the language of all Boolean formulas F for which more than half of all assignments satisfy F. You should be able to easily show that MAJ-SAT is in PP. In fact, it is also very easy to show that MAJ-SAT is "PP-complete". In particular, MAJ-SAT is in P if and only if P = PP.

The weird nature of PP.

On one hand, PP contains both NP and coNP. On the other hand, it is not known to contain the class P^{NP} (if you remember what that is: namely, the languages decidable by polynomial time algorithms with access to a SAT oracle). On the other other hand, it is known that the entire "polynomial hierarchy" (a vast generalization of NP, which is "almost" as large as PSPACE) is contained in P^{PP} (i.e., polynomial-time algorithms with access to

an oracle for MAJ-SAT). This is "Toda's Theorem". So yeah, PP is kind of a weird and unnatural class; the basic version is not very powerful; it's not closed under "subroutines"; but when you "close" it under subroutines it becomes super-powerful.

References

[Aar10] S. Aaronson. BQP and the Polynomial Hierarchy. *Proceedings of ACM STOC 2010*, pages 141–150, 2010.

[ADH97] L. Adleman, J. DeMarrais, and M. Huang. Quantum Computability. *SIAM Journal on Computing*, pages 1524–1540, October 1997.

[AKS04] M. Agrawal, N. Kayal, and N. Saxena. PRIMES is in P. *Annals of Mathematics 160*, pages 781–793, 2004.

[BV97] E. Bernstein and U. Vazirani. Quantum Complexity Theory. *SIAM Journal on Computing, Volume 26*, pages 1411–1473, October 1997.

[DN05] C. Dawson and M. Nielsen. The Solovay-Kitaev Algorithm. *arXiv:quant-ph/0505030v2*, August 2005.