MP 472 Quantum Information Processing

http://www.thphys.may.ie/staff/jvala/MP472.htm

Outline:

Deterministic Turing machine
- definition
- example: unary addition

Computability
- computable functions
- decidable predicates
- universal Turing machine

Computational complexity
- time and space
- P and PSPACE
- Non-deterministic computation and NP, NP hardness, NP completeness
- Probabilistic computation and BPP
- Quantum computation and BQP

Deterministic computation and deterministic Turing machine

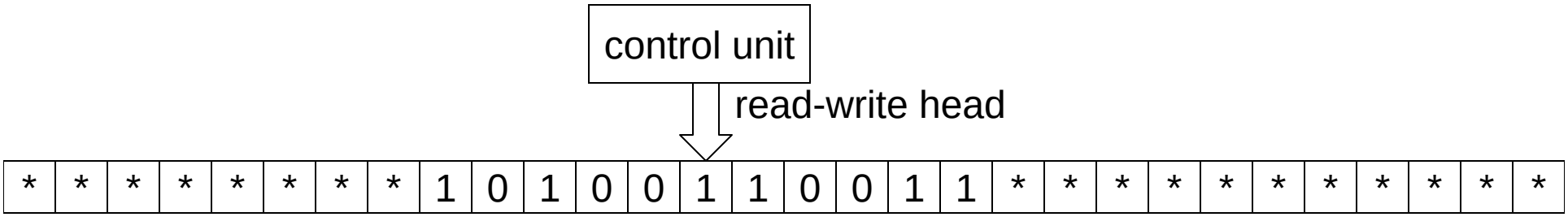
TM consists of:

i) a finite *alphabet* Σ containing the blank symbol *

ii) a 2-way infinite *tape* divided into cells, one of which is a special *starting cell*. Each cell contains a symbol from the alphabet Σ. All but a finite number of the cells contain the special blank symbol *, denoting an empty cell;

iii) *read-write head* that examines a single cell at a time and can move left ($\leftarrow$) or right ($\rightarrow$);

iv) a *control unit* along with a finite set of states Γ including a distinguished *starting state*, γ_0, and a set of *halting states*.



The computation of a TM is controlled by a *transition function*:

$$\delta: \Gamma \times \Sigma \longrightarrow \Gamma \times \Sigma \times \{\leftarrow, \rightarrow\}$$

Example: unary addition TM

States: $\Gamma = \{\gamma_0, \gamma_1, \gamma_2, \gamma_3\}$ with the starting state γ_0 and the halting state γ_3;

Alphabet: $\Sigma = \{*, 1, +, =\} = \Sigma_0 \cup \{*\}$, where Σ_0 is called an external alphabet;

Input: integers $a, b \geqq 0$ with the symbols + and =
(e.g. 2+1 is written as '11 + 1 =' on the tape with the leftmost input symbol in the starting square)

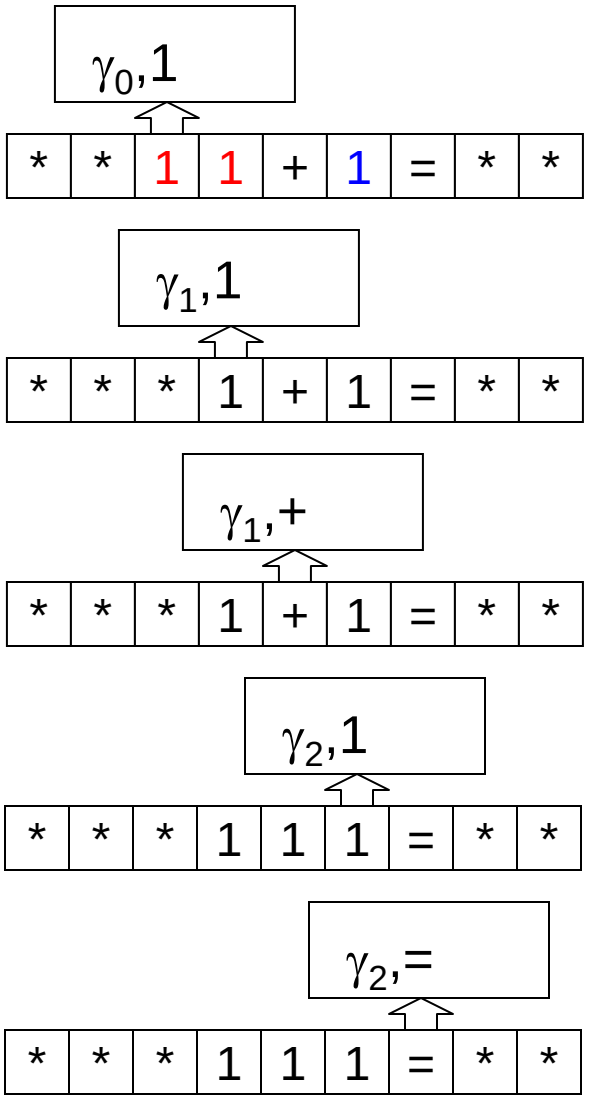
Output: a + b unary

Transition function:

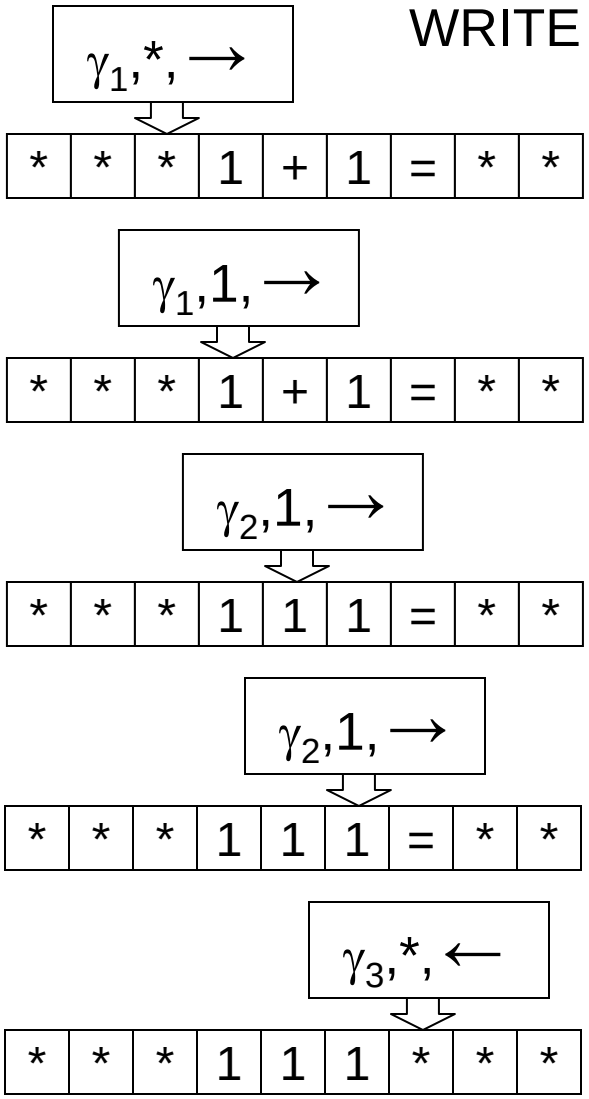
$(\gamma_0, 1, \gamma_1, *, \rightarrow)$	$a \neq 0$, reading a
$(\gamma_0, +, \gamma_2, *, \rightarrow)$	$a = 0$, erase + read b
$(\gamma_1, 1, \gamma_1, 1, \rightarrow)$	reading a
$(\gamma_1, +, \gamma_2, 1, \rightarrow)$	replace + by 1 read b
$(\gamma_2, =, \gamma_3, *, \leftarrow)$	finish reading b, erase = halt

Example: 2+1 unary

READ



WRITE



$(\gamma_0,1, \gamma_1,*, \rightarrow)$

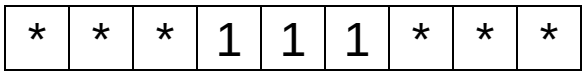
$(\gamma_0,+, \gamma_2,*, \rightarrow)$

$(\gamma_1,1, \gamma_1,1, \rightarrow)$

$(\gamma_1,+, \gamma_2,1, \rightarrow)$

$(\gamma_2,=, \gamma_3,*, \leftarrow)$

result



Computable functions and decidable predicates

Every TM computes a partial function φ_M: $\Sigma_0^* \rightarrow \Sigma_0^*$, where Σ_0^* is the set of all strings over Σ_0 (external alphabet). $\varphi_M(x)$ is the output string for input x.

A partial function *f*: $\Sigma_0^* \rightarrow \Sigma_0^*$ is *computable* if there exist a TM s.t. $\varphi_M = f$.

A predicate is a function *L*: $\Sigma_0^* \rightarrow \{0,1\}$ (a function with a Boolean value).
A predicate is called *decidable* if this function is computable.

Subsets of Σ_0^* are also called languages over Σ_0.

Church-Turing thesis: Any algorithm can be realized by a Turing machine.

TM is a finite object so it can be encoded by a string. Then for any fixed alphabet Σ_o^*, we can consider a universal Turing machine U which computes the function $u([M],x) = \varphi_M(x)$ where [M] is the encoding of TM.


Computational complexity

Complexity

Note this is the worst case

A TM works in time T(n) if it performs at most T(n) steps for any input of size n.

A function (predicate) F on $\mathbb{B}^*$, i.e. on binary strings, is *computable (decidable) in polynomial time* if there exist a TM that computes it in time T(n) = poly(n), where n is the input length
(poly(n) means polynomial growth, i.e. poly(n) $\leq cn^d$ for some constants c and d and sufficiently large n).

A class of all functions (predicates) computable (decidable) in polynomial time is called P.

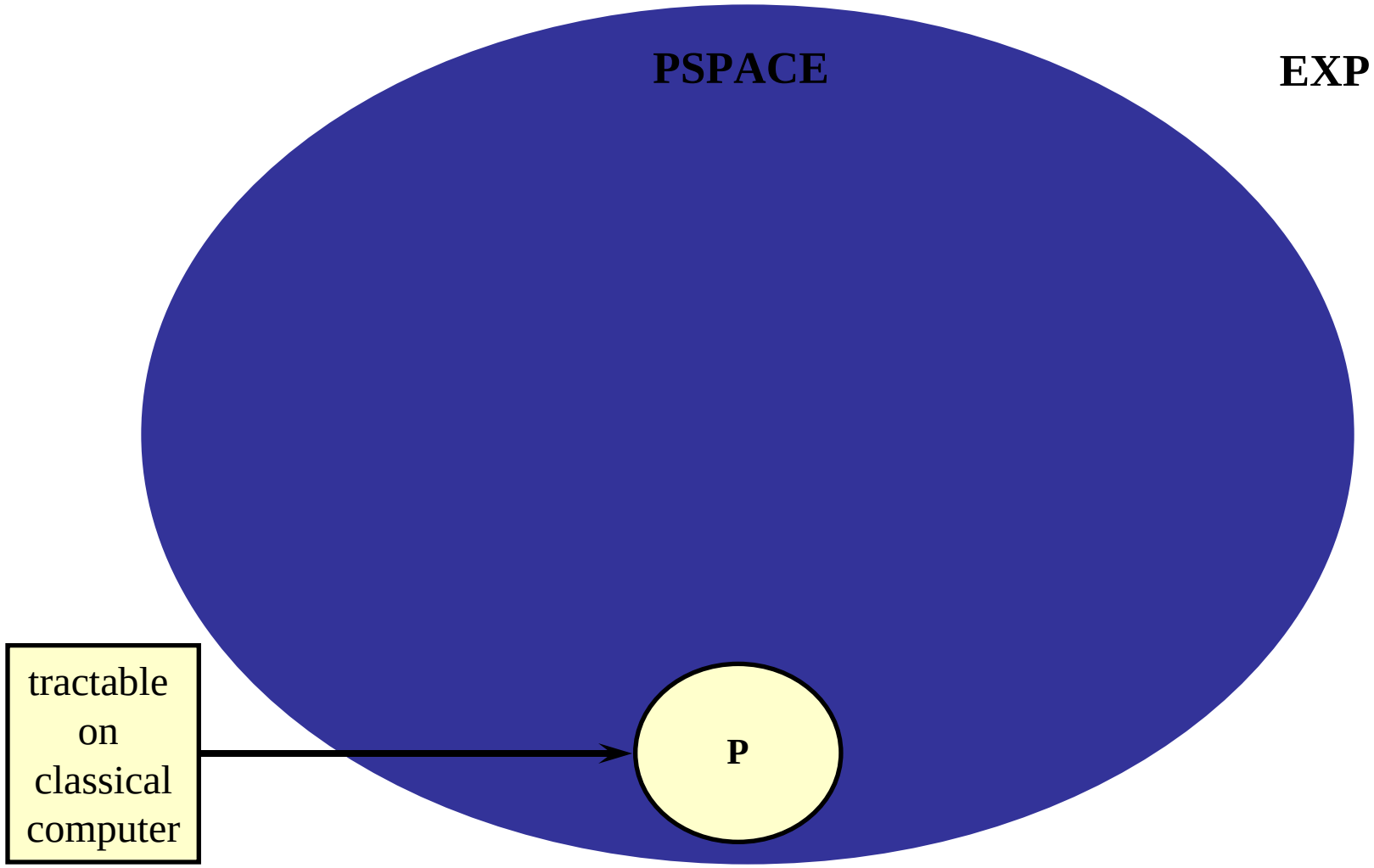
A TM works in space s(n) if it visits at most s(n) cells for any computation on inputs of size n.

A function (predicate) F on $\mathbb{B}^*$, is *computable in polynomial space* if there exist a TM that computes F and runs in space s(n) = poly(n), where n is the input length.

A class of all functions (predicates) computable (decidable) in polynomial space is called PSPACE.

P ⊆ PSPACE

Though this is an open question (your chance!), it is generally believed that this inclusion is strict, i.e. P ≠ PSPACE:



Non-deterministic computation

A non-deterministic TM is a hypothetical machine that resembles deterministic TM, but can non-deterministically choose one of several actions possible in a given configuration, i.e. a transition function of NTM is multi-valued.

A predicate *L* belongs to the class NP (Non-deterministic Polynomial) if there exist a non-deterministic Turing machine M and a polynomial p(n) s.t.

L(x) = 1 ⇒ there exist a computational path that gives answer "yes" in time in time p(|x|), where |x| is the size of x;

L(x) = 0 ⇒ there is no path with this property.

Alternative definition of NP (Kitaev):

Imagine two persons: King Arthur (with polynomially bounded mental capabilities) and a wizard Merlin (intelectually omnipotent). Arthur is interested in *L*(x). Merlin wants to convince Arthur that *L*(x) is true, but Arthur does not trust Merlin and wants to make sure *L*(x) is true. So Arthur arranges that, after both he and Merlin see input string x, Merlin writes a note to Arthur where he proves that *L*(x) is true. Then Arthur verifies this proof by some polynomial proof-checking predicate (procedure), *R*(x,y) = "y is a proof of *L*(x)". It should satisfy:

L(x) = 1 ⇒ Merlin can convince Arthur that *L*(x) is true by presenting some proof y s.t. *R*(x,y);

L(x) = 0 ⇒ whatever Merlin says Arthur is not convinced: *R*(x,y) is always false

NP, NP hardness and NP completeness

A predicate L_1 is *reducible* to a predicate L_2 if there exists a function $f \in P$ such that $L_1(x) = L_2(f(x))$ for any input string x. We say $L_1 \propto L_2$.

Karp reducibility

Lemma: Let $L_1 \propto L_2$, then

(a) $L_2 \in P \Rightarrow L_1 \in P$

(b) $L_2 \notin P \Rightarrow L_1 \notin P$

(c) $L_2 \in NP \Rightarrow L_1 \in NP$

A predicate *L* is NP-hard if any predicate in NP is reducible to it.

A predicate *L* is NP-complete if it is NP-hard and $L \in NP$.

Example: SAT (satisfiability)

SAT(x) means x is a propositional formula (containing Boolean variables and operations (negation, disjunction, conjunction) that is satisfiable, i.e. “true” for some values of the variables.

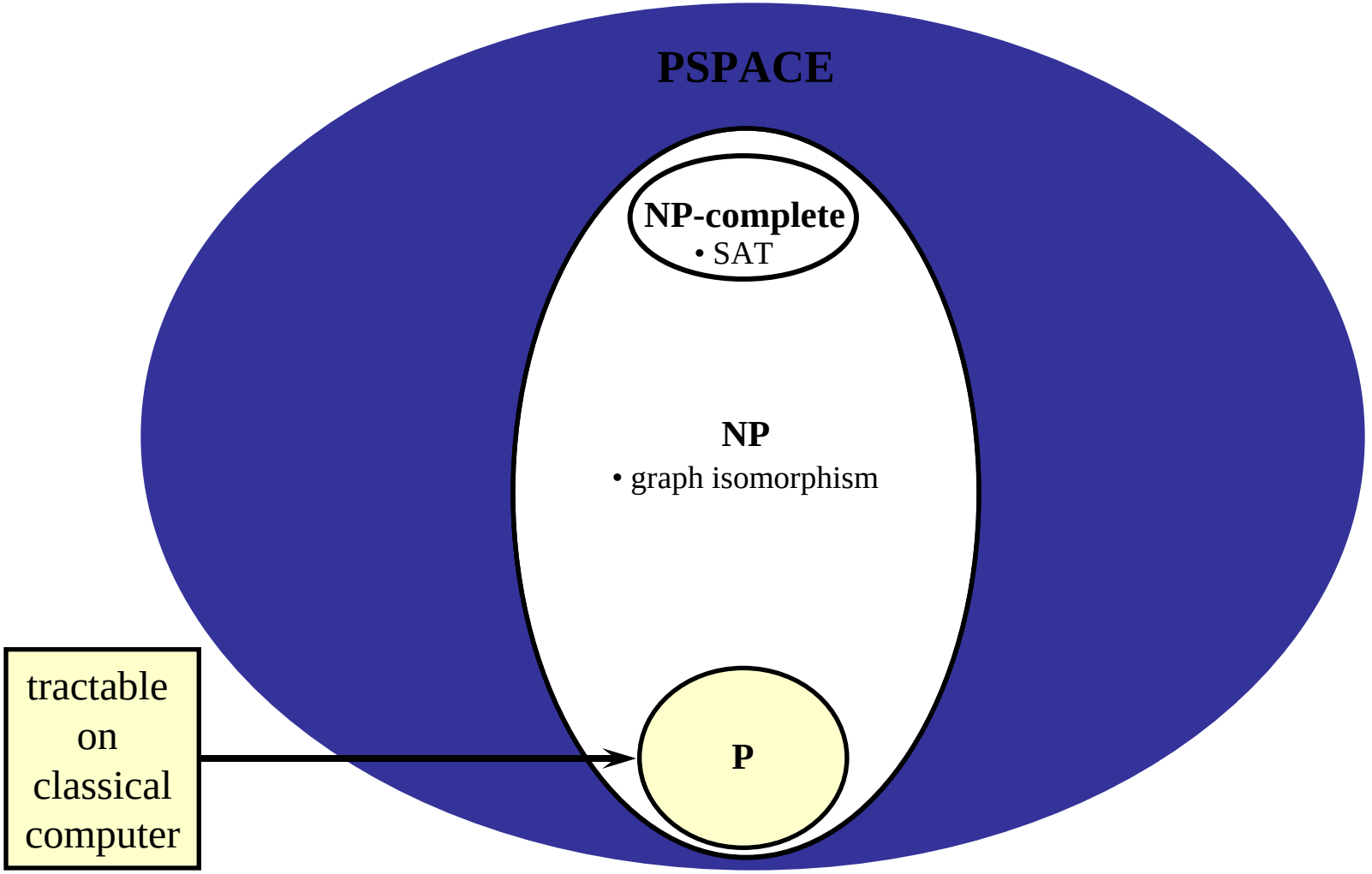
Cook-Levin Theorem: $SAT \in NP$;
$SAT \in NP\text{-complete}$.

Other examples: 3-COLORING, CLIQUE, …

P ⊆ NP ⊆ PSPACE

Though these are open questions (your chances!), it is generally believed that these inclusions are strict, i.e. P ≠ NP and NP ≠ PSPACE:

(Imagine that you would prove that SAT ∈ P, then you would resolve the problem P vs. NP which is one of the Millennium problems of the Clay Mathematics Institute with a prize of $1,000,000 !!!).



Probabilistic computation

A probabilistic Turing machine can probabilistically choose one of several actions possible in a given configuration (note that this is somewhat similar to a non-deterministic TM but the choice is made by coin tossing rather than guessing; PTM is more than hypothetical - it is physical !!!).

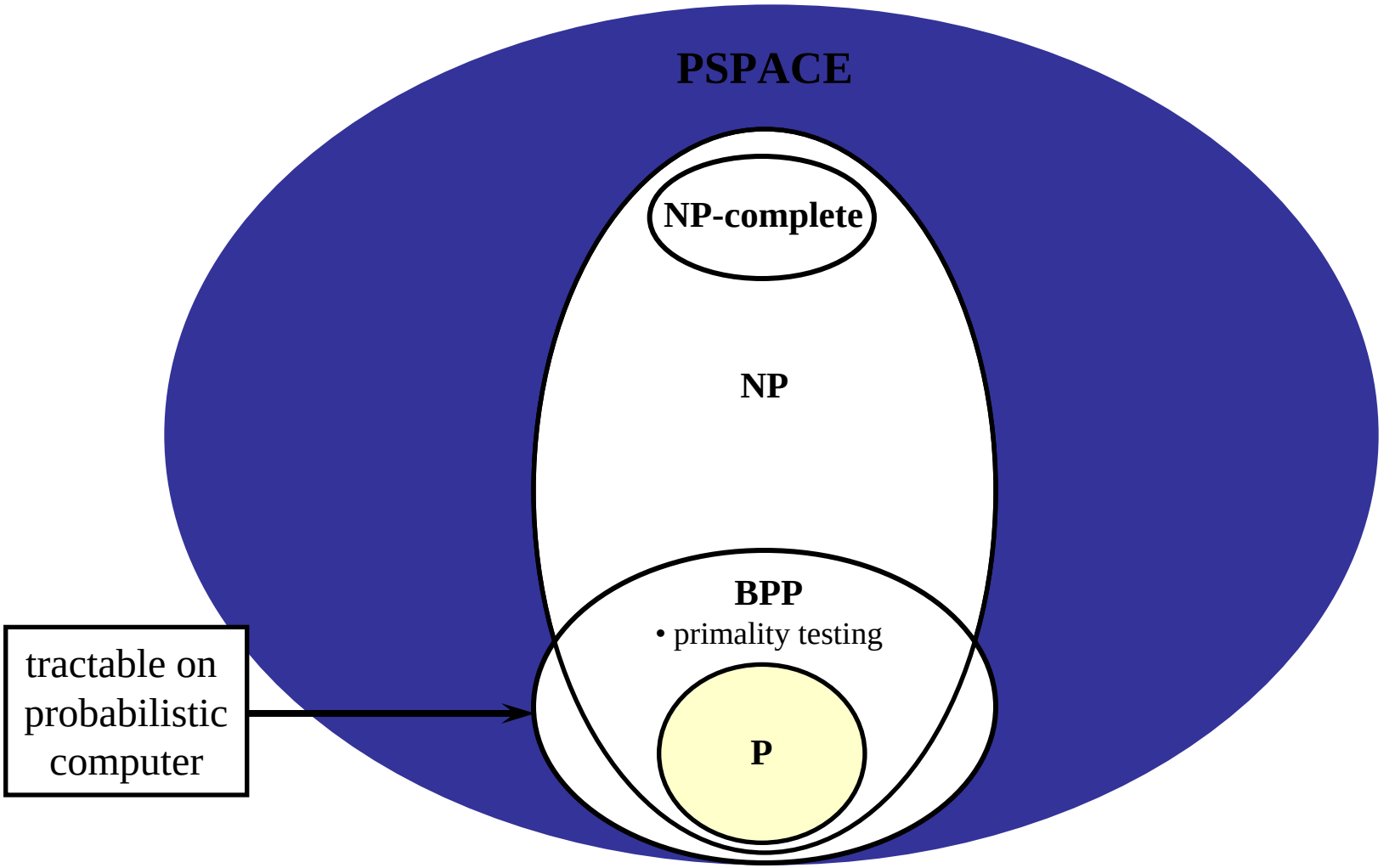
Let ε be a constant such that $0 < \varepsilon < ½$. A predicate *L* belongs to the class BPP (Bounded-Error Probabilistic Polynomial) if there exist a probabilistic Turing machine M and a polynomial p(n) s.t. the machine M running on input string x always terminates at most p(|x|) steps, and

$L(x) = 1 \Rightarrow$ M gives the answer "yes" with probability $\geq 1-\varepsilon$;

$L(x) = 0 \Rightarrow$ M gives the answer "no" with probability $\leq \varepsilon$.

Example: PRIMALITY, i.e. checking whether a given integer is a prime number.

P ⊆ BPP ⊆ PSPACE


PSPACE
NP-complete
NP
BPP
• primality testing
P
tractable on probabilistic computer

Quantum computation

A quantum Turing machine can choose a superposition of several actions in a given configuration (note that this is somewhat similar to a probabilistic TM).

Let ε be a constant such that $0 < \varepsilon < ½$. A predicate *L* belongs to the class BQP (Bounded-Error Quantum Polynomial) if there exist a quantum Turing machine M and a polynomial p(n) s.t. the machine M running on input string x always terminates at most p(|x|) steps, and

$L(x) = 1 \Rightarrow$ M gives the answer "yes" with probability $\geq 1-\varepsilon$;

$L(x) = 0 \Rightarrow$ M gives the answer "no" with probability $\leq \varepsilon$.

Alternatively (using the concept of quantum circuit):

A *quantum algorithm* for the computation of a function $F: \mathbb{B}^* \rightarrow \mathbb{B}^*$ is a classical algorithm (i.e. a Turing machine) that computes a function of the form $x \mapsto Z(x)$, where Z(x) is a description of a quantum circuit which computes *F*(x) on empty input.

The function *F* is said to belong to the class BQP if there is a quantum algorithm that computes *F* in time poly(n).

P ⊆ BPP ⊆ BQP ⊆ PSPACE