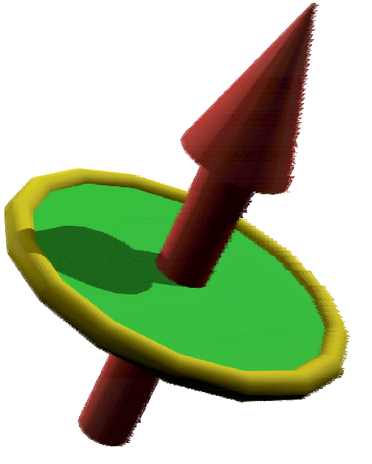
Review of Computer Science related to Quantum Computing

Sources:

Chuang and Nielsen, Vadim Bulitko, Michele Mosca, Artur Ekert,

Joost N. Kok, Petros Koumoutsakos & Bernt Schiele,

Thomas Werder & Bastian Leibe



Artificial Intelligence

- On May 11, 1997, an IBM computer named Deep Blue whipped world chess champion Garry Kasparov in the deciding game of a six-game match

What is Artificial Intelligence?

- ***Variant 1.*** The concept that machines can be improved to assume some capabilities normally thought to be like human intelligence such as **learning, adapting, self-correction, etc.**
- ***Variant 2.*** The extension of human intelligence through the use of computers, as in past times the **physical power was extended** through the use of mechanical tools.

- ***Variant 3.*** Movie Artificial Intelligence by Steven Spielberg

Artificial Intelligence

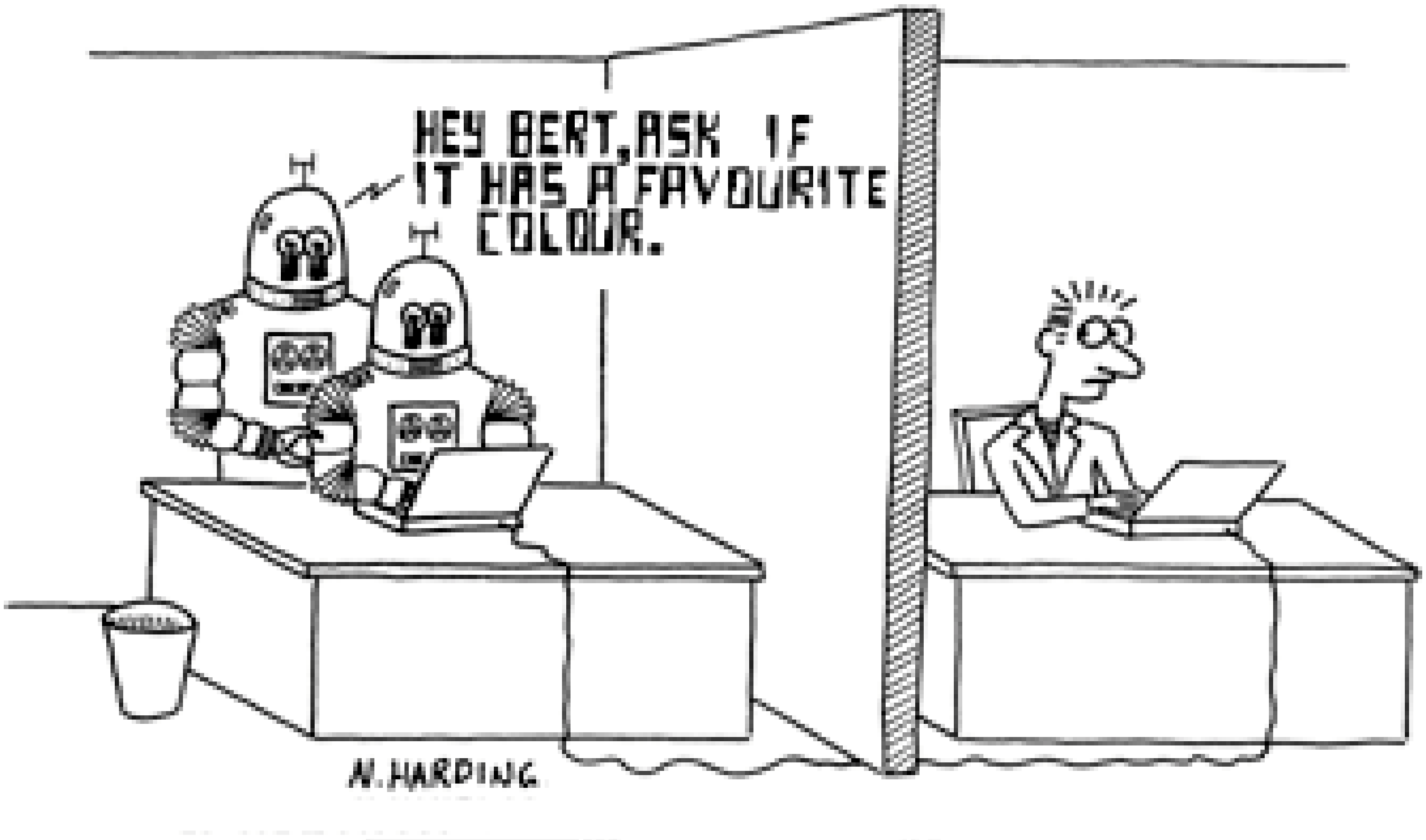
- First Robot World Cup Soccer Games held in Nagoya, Japan in 1997
- Goal: team of robots beats the FIFA World Cup champion in 2050

Artificial Intelligence

- Alan Turing
- Turing Award
- Turing Machine
- Turing Test

Artificial Intelligence

- Turing Test



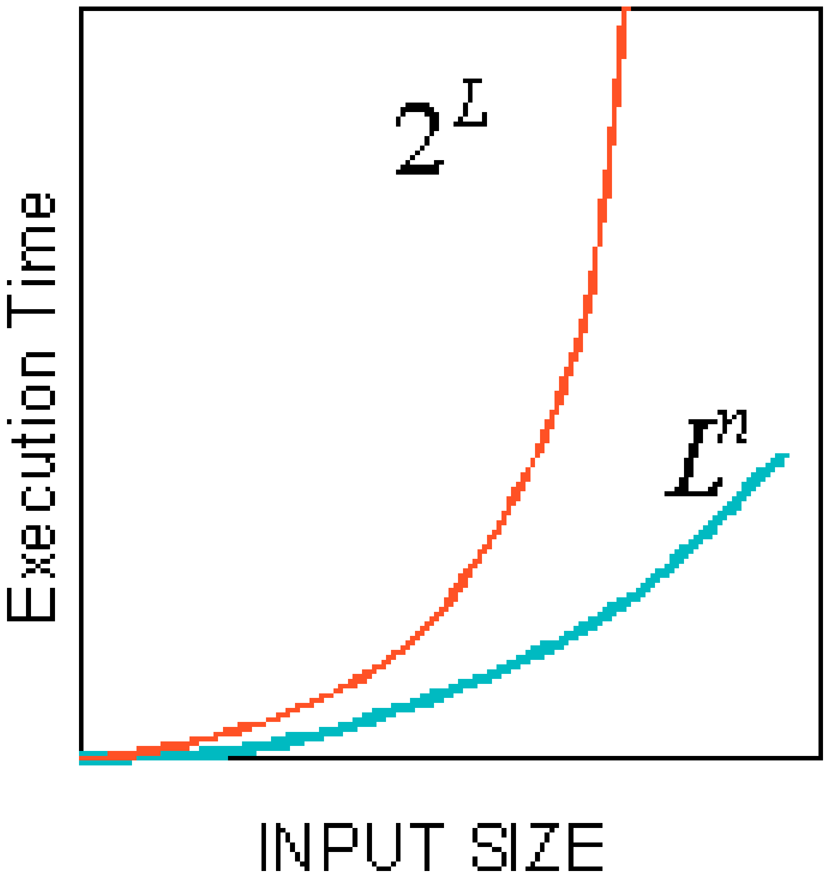
Artificial Intelligence

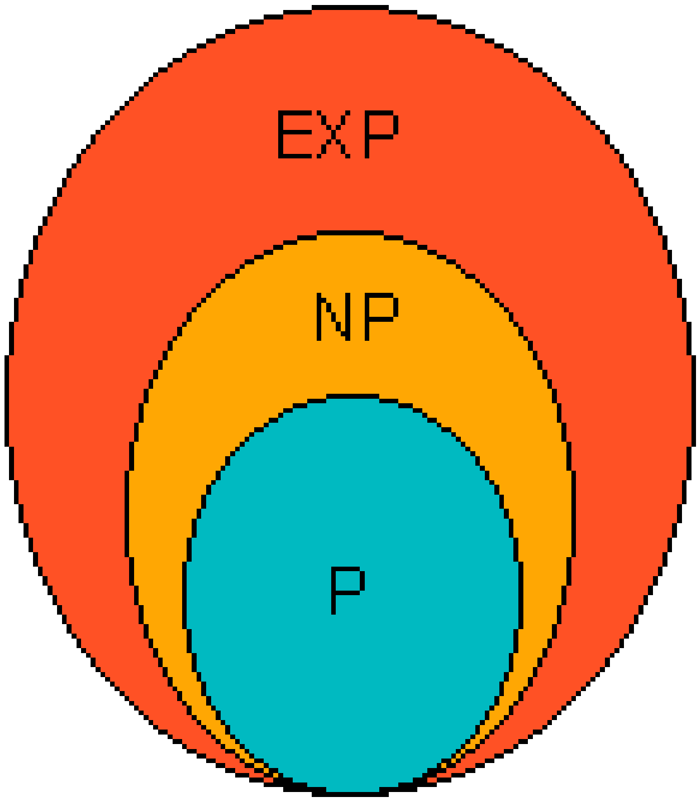
- Natural language processing: it needs to be able to communicate in a natural language like English
- Knowledge representation: it needs to be able to have knowledge and to store it somewhere
- Automated reasoning: it needs to be able to do reasoning based on the stored knowledge
- Machine learning: it needs to be able to learn from its environment

Time Complexity

- Turing machine gives notion of computability
- Time complexity: how many steps does it take to find an answer?
- Combinatorial Explosion
- Problems that are computable in polynomial time (class P)
- Problems that are verifiable in polynomial time (class NP)
- **P equals NP?**

Computational Complexity





Which problems are hard?

Addition

```
 123
 654
---- +
 777
```

Multiplication

```
  123
  654
----- *
  492
 615.
738..
----- +
80442
```

Factoring

```
943=?*?
```

```
943=23*41
```

EASY **HARD**

Common sense

- In practice an algorithm that solves a problem with $2^{n/1000}$ operations is probably more useful than an algorithm that solves this problem in n^{1000} operations.

FACTORING is tough

- factor number N of L decimal digits

$$N \approx 10^L$$

- # divisions required (approximately)

$$\sqrt{N} \approx 10^{L/2}$$

- exponential in L

Factoring - example

- Try to factor L=100 digit number with the trial division method at 10^6 divisions per second
- Number of divisions $\approx 10^{50}$
- Execution time $\approx 10^{44}$ seconds
- Age of the Universe $\approx 10^{17}$ seconds
- Applications…

Mathematically solvable
versus physically solvable...

Computer Science

- Goal: *Brush up on CS aspects relevant to QC*
- Models of computation:
  - Turing machines
  - Circuits
- Computation problems
  - Description
  - Algorithms
  - **Complexity** : asymptotic notation
  - **Complexity** : classes
- Energy & computation : reversibility

Models Of Computation

- Why do we need a model of computation?
- When someone says *"This function is incomputable"* or *"f(x) is computable but intractable"*, etc. **what does it really mean?**
- What if I say *"I can compute this"* or *"I have an algorithm for this"* ?
- **Intuition**?
- Well, David Hilbert felt that any true formula can be proven by a ***mechanical procedure***.
- …Need a formalization

Now we go back to mathematics for a while...

The Hilbert Problems

- The German mathematician David Hilbert (1862-1943) was born on Jan. 23, 1862, in Konigsberg, Prussia (now Kaliningrad, Russia). He received his doctorate from the University of Konigsberg in 1884 and remained there as a professor from 1886 to 1895. In 1895 he joined the University of Gottingen and retired in 1930.
- Hilbert reduced Euclidean geometry to a series of axioms.
- A substantial part of Hilbert's fame rests on a list of 23 research problems he presented in 1900 to the International Mathematical Congress in Paris.
- He surveyed nearly all the mathematics of his day and set forth the problems he thought would be significant for mathematicians in the 20th century.

The Hilbert Problems contd.

- Many of the problems have since been solved, and each solution was a noted event.
- ***Hilbert second problem*** asked **whether it can be proved that that the axioms of arithmetic are consistent** (that is, that a finite number of logical steps based on them can never lead to contradictory results).
- **Godel's solution**: you cannot tell, because propositions can be formulated that are ***undecidable*** within the axioms of arithmetic!
- Example: By definition, x^2=x.x, x^3=x.x.x, and so on.
- Now what does $x^{3/5}$ mean?
- Can we be certain that the meanings that we have given to fractional and non-rational exponents are always consistent with the natural meaning of positive integral exponents?
- That is the nature of Hilbert second problem

Kurt Godel (1906-1978)

- In 1931 the Austrian mathematician and logician Kurt Godel published what has been called **Godel's proof** in arithmetic.

- This proof states that within any rigidly logical mathematical system there are propositions (or statements) that *cannot be proved or disproved* on the basis of the axioms within that system.
- It is therefore uncertain that the basic axioms of arithmetic will not give rise to contradictions.

- This proof has become a **hallmark of 20th-century mathematics**, and its significance is still debated.

Gödel's incompleteness theorem (1931):

- **Any system of logic powerful enough to express elementary arithmetic contains true statements that cannot be proven within that system.**
- The halting problem is the typical example for a problem that is ***not decidable*** (not computable, not solvable by a TM).
- Many problems can be shown to be undecidable by reducing them to the halting problem.

Implications of Goedel's Theorem

- **"It appears to foredoom the ideal of science - devising a set of axioms from which all phenomena of the natural world can be deduced"**
  - **Carl Boyer in A History of Mathematics**

Is a quantum computer fundamentally stronger than classic one on undecidable problems - **no.**

Can a quantum computer solve these problems more efficiently than any existing computer - **yes**

Thus mathematics proves that no humans nor computers can have any theoretical advantages to create future mathematics better than the other and the "universal mathematician's *philosophy stone* does not exist".

Relevant Sources on math review for those interested…..

- 1. Compton’s Interactive Encyclopedia (1995)
- 2. Boyer, C. B. A History of Mathematics, Second Edition, John Wiley & Sons (1991), p.611
- 3. “Time” special issue on “Scientists & Thinkers of the 20th Century”, March 29, 1999, Vol 153 No. 12, pp. 64-205
- 4. http://www.clarku.edu/~hmarek/html/godel.html (This is an instructive place on www)
- 5. http://www.clarku.edu/~hmarek/html/disc.html

Turing Machines

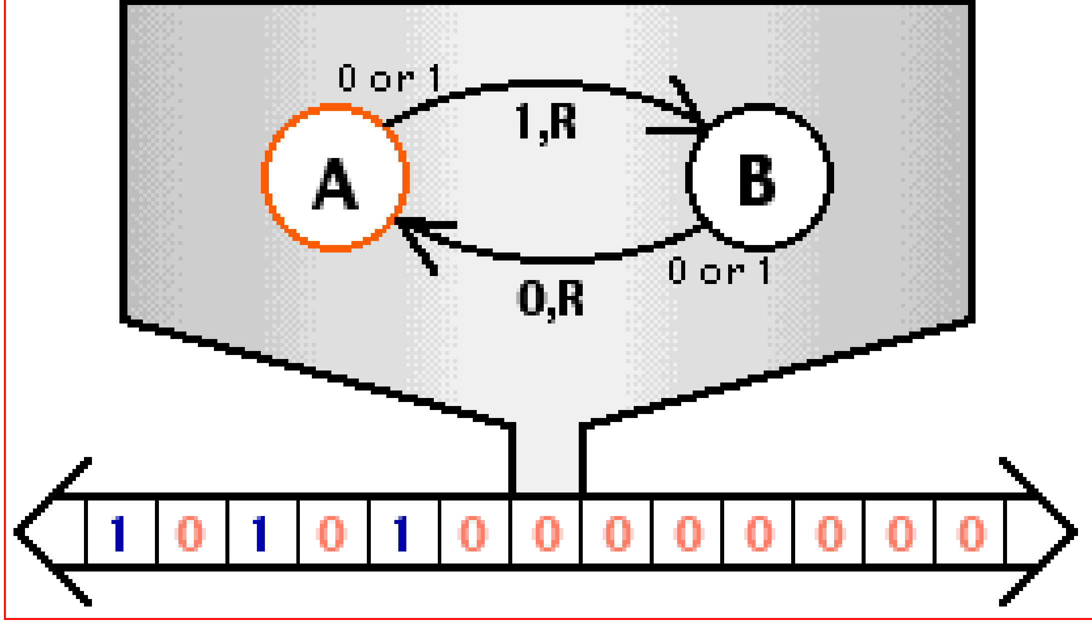
- Need a formalization of what it means to **have an algorithm for** (or to be able to compute)

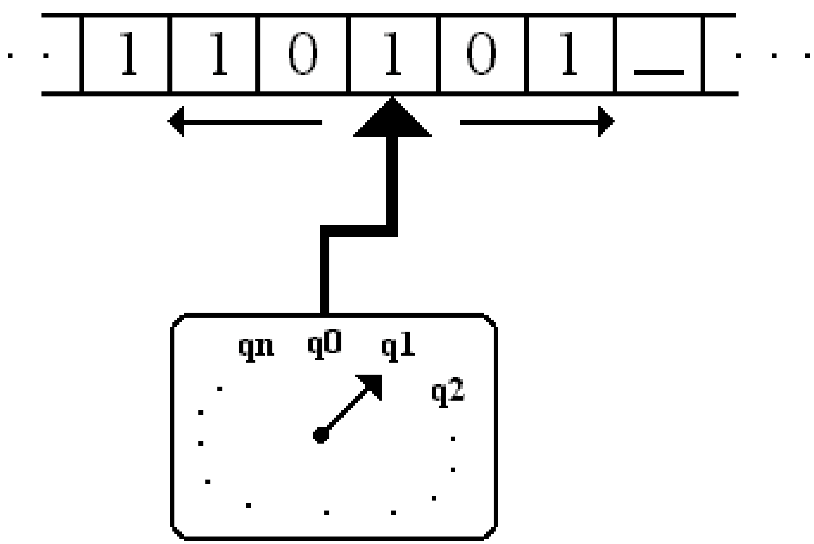
LCMs can do anything that could be described as 'rule of thumb' or 'purely mechanical'.

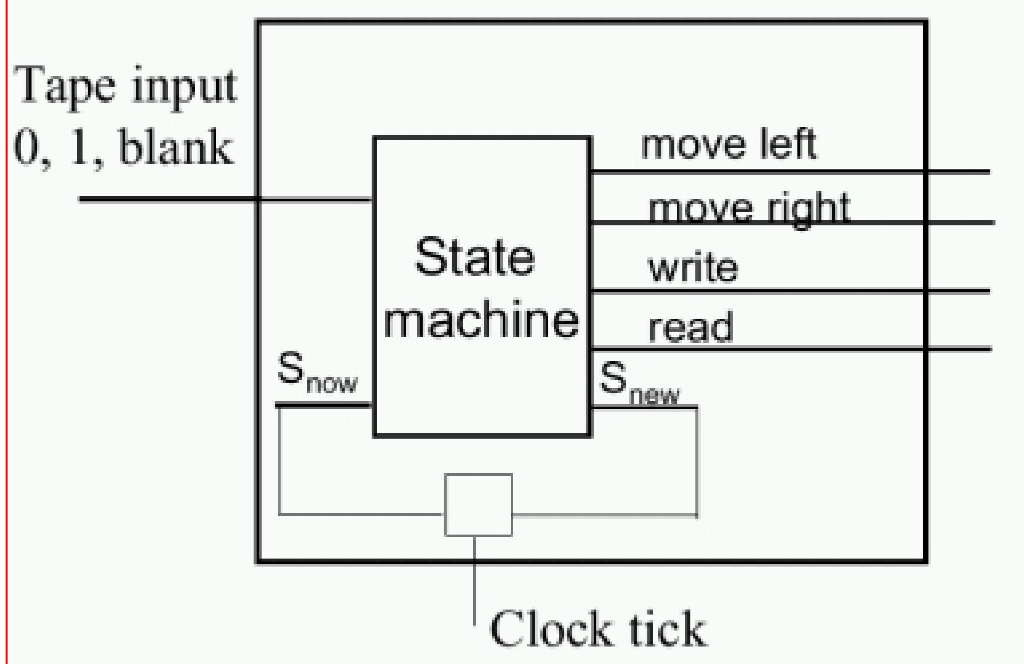
Alan Mathison Turing (1948)

- So what is **LCM** (or as it's now known **Turing** Machine) ?

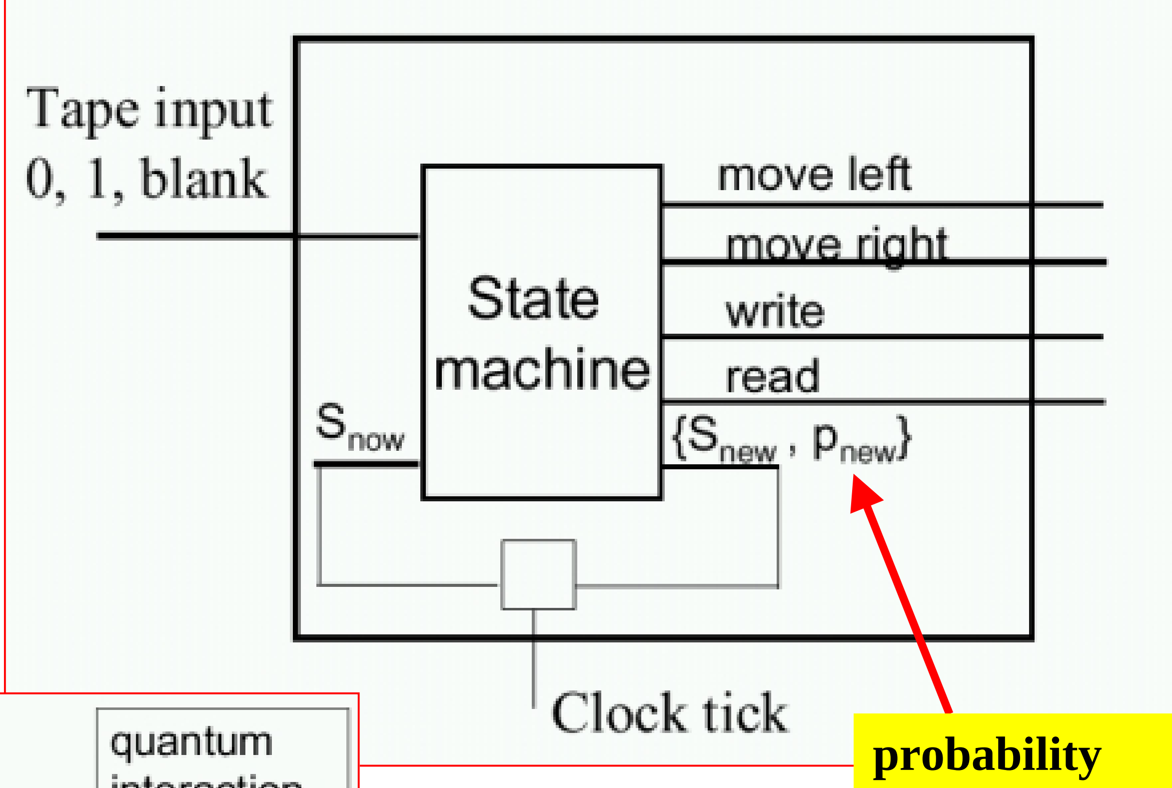
Turing Machines



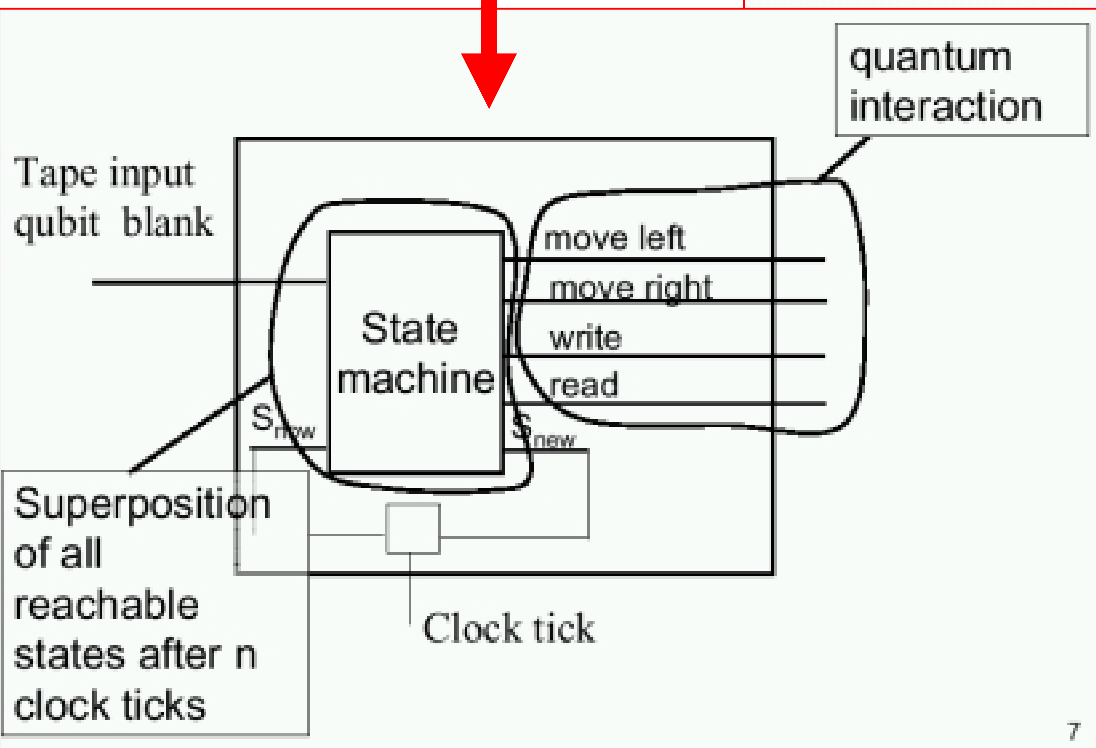




Probabilistic Turing Machine



Quantum Turing Machine



It was also discussed in the lecture how to build a non-deterministic Turing Machine and other types of quantum Turing Machines

Turing machines

- Finite state control: consists of finite set of internal states $q_1, \ldots, q_m$ and special states q_s and q_h which are the starting state and halting state, respectively.
- Tape: one-dimensional object which stretches to infinity in one direction and consists of the tape squares. The tape squares each contain one symbol drawn from some alphabet ***A***.
- Read/Write Head: identifies a square on the tape which is being accessed by a machine.
- Program: finite ordered list of program lines in the form:

$$< q, x, q'_0, x'_0, s_i >.$$

Turing machine: How does it work?

- The Turing machine looks through the lines of the program searching for a line **<q,x….>,** where **q** is the current state of the machine and **x** is the symbol being read on a tape.
- If it can find such a line it changes the state of the machine to **q'**, overwrites the symbol on the tape square to **x'** and moves the read/write head by **s** tape squares.
- If it can not find such a line the internal state of the machine is changed to $\mathbf{q_h}$, machine halts operation and whatever is one the tape is an output.

Turing machine: Example

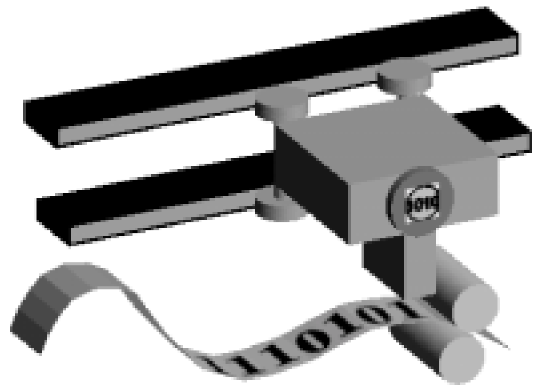
- Internal states: q_1; q_2; q_3; q_h; q_s.
- Alphabet: $\triangleright$(marks left hand edge), 0, 1 and blank spaces (designated in program as b).
- Tape initially contains binary number x followed by all blanks.
- Program:

$$\langle q_s, \triangleright, q_1, \triangleright, +1\rangle$$
$$\langle q_1, 0, q_1, b, +1\rangle$$
$$\langle q_1, 1, q_1, b, +1\rangle$$
$$\langle q_1, b, q_2, b, -1\rangle$$
$$\langle q_2, b, q_2, b, -1\rangle$$
$$\langle q_2, \triangleright, q_3, \triangleright, +1\rangle$$
$$\langle q_3, b, q_h, 1, 0\rangle$$

What does it do?

It computes f(x) = 1.

Universal computation.



- Turing machines.
  - See R. Penrose, *The Emperor's New Mind,* page 71.
- Church-Turing thesis:

  A computable function is one that is computable by a universal Turing machine.

Church-Turing thesis:

- The class of functions computable by a Turing machine corresponds exactly to the class of functions which we would naturally regard as being computable by an algorithm.
- The thesis asserts equivalence between a rigorous mathematical concept, i.e. function computable by the Turing machine and the intuitive concept what it means for a function to be computable by an algorithm. No evidence to the contrary has been found.
- Quantum computers also obey Turing thesis, the difference is in efficiency.
- Different versions of the Turing machine: multi-tape machines, introduction of the randomness in the model.

Demos of Turing Machines

- Classical implementation

  http://www.warthman.com/ex-turing.htm

- Conway's Game of Life implementation

  http://www.rendell.uk.co/gol/tmdetails.htm

Universal Turing Machines

- A Turing Machine (T M) is a finite state machine with a tape of unbounded length.
- A function F(x) is ***Turing computabl**e* if a TM exists which, if fed with x, will eventually halt and write F(x) on the tape.
- A Universal Turing Machine (UTM) is capable of ***imitating (simulating) any other*** given TM.
  - Several ways of constructing such machines were given, first by Turing. See on WWW.

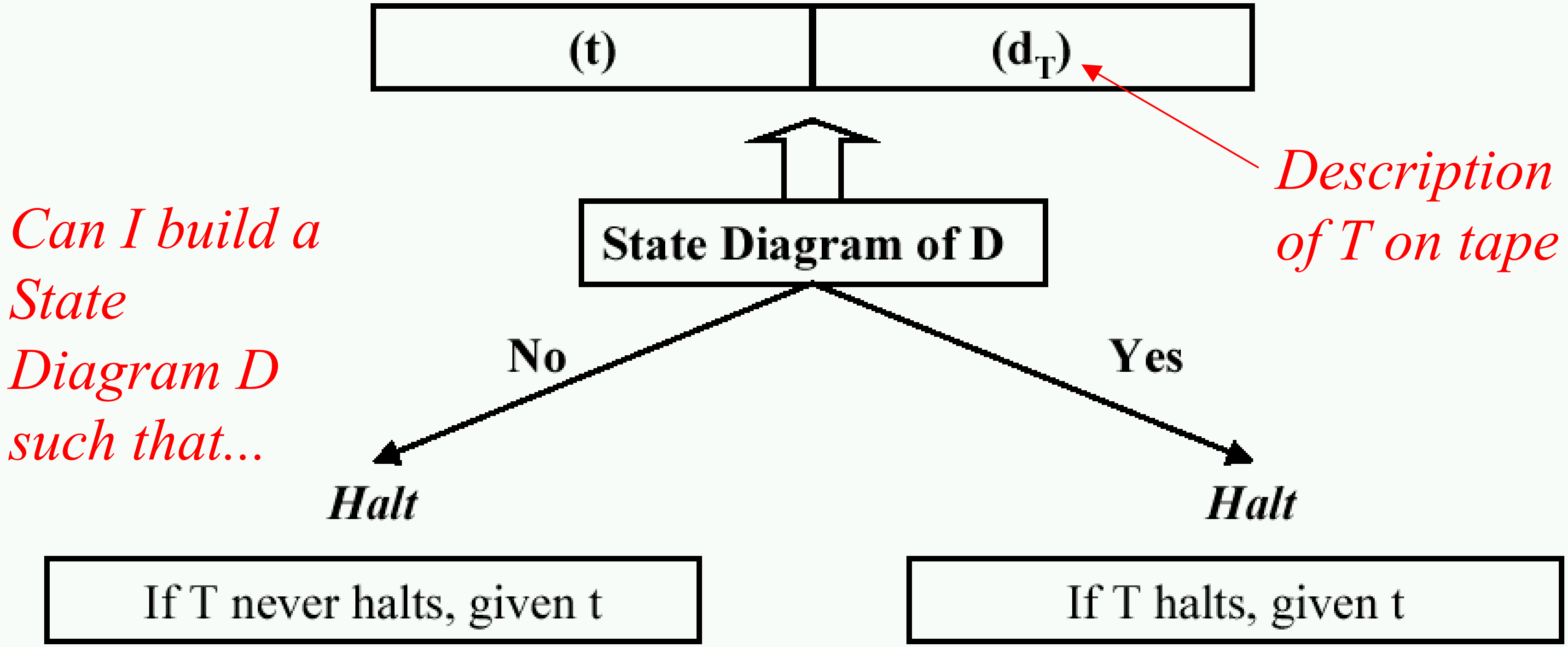
The halting problem

- **Question:** *Is it possible to build a machine that will tell us whether a Turing machine T* ***with tape t*** *will halt?*
- **Answer:** *No, this is not possible!*
  - 1. What is a Turing Machine (*More precise* r*epetition?*)
  - 2. Reformulate the halting problem more precisely
  - 3. Discuss the Importance of the halting problem

- ***Literature:*** R. Feynman, *Lectures on Computation,* Penguin Books 1996

The Halting Problem reformulated

- Definition: Let **D** be a UTM with the added property that *it tells us whether or not* ***T*** *with tape* **t** *halts*.
- Question: Does a machine **D** exist?

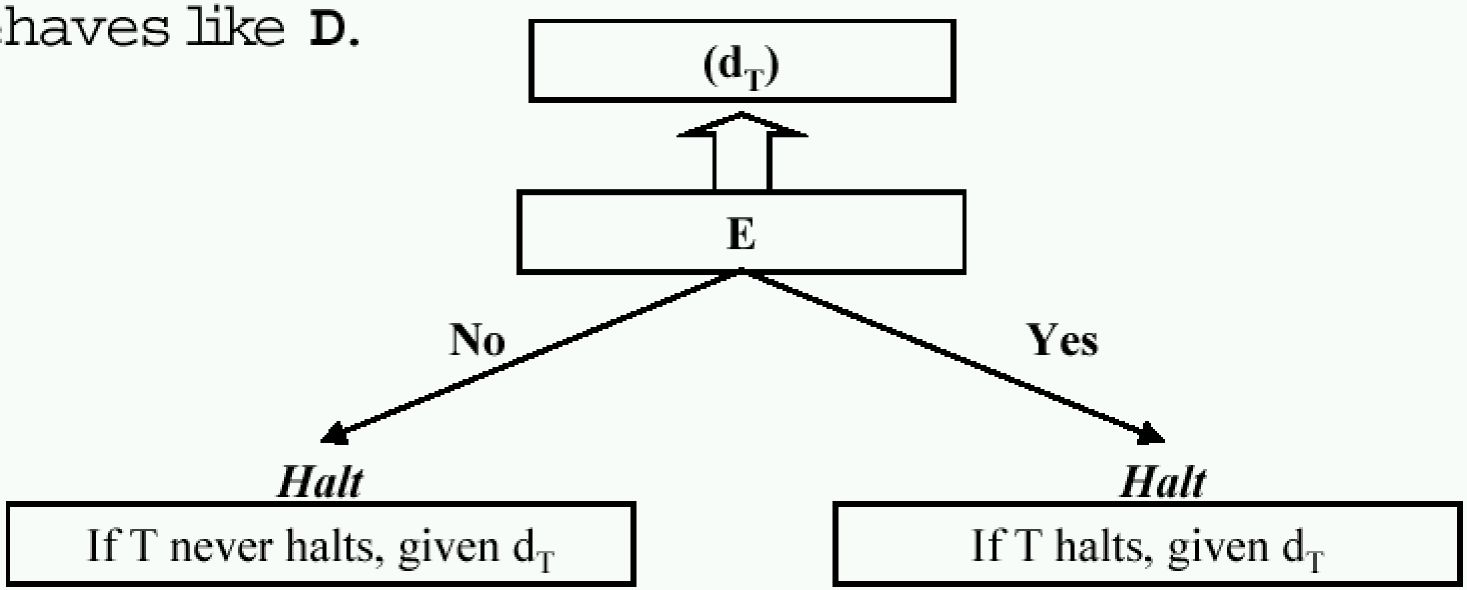


- *D=UTM*
- *T=TM*

Proof (cont)

We will prove that the halting problem cannot be solved by a TM by constructing a contradiction.

Step 1: Introduce a machine **E**, that reads a tape $\mathbf{d_T}$, copies it onto a blank part of the tape, and then behaves like **D**.

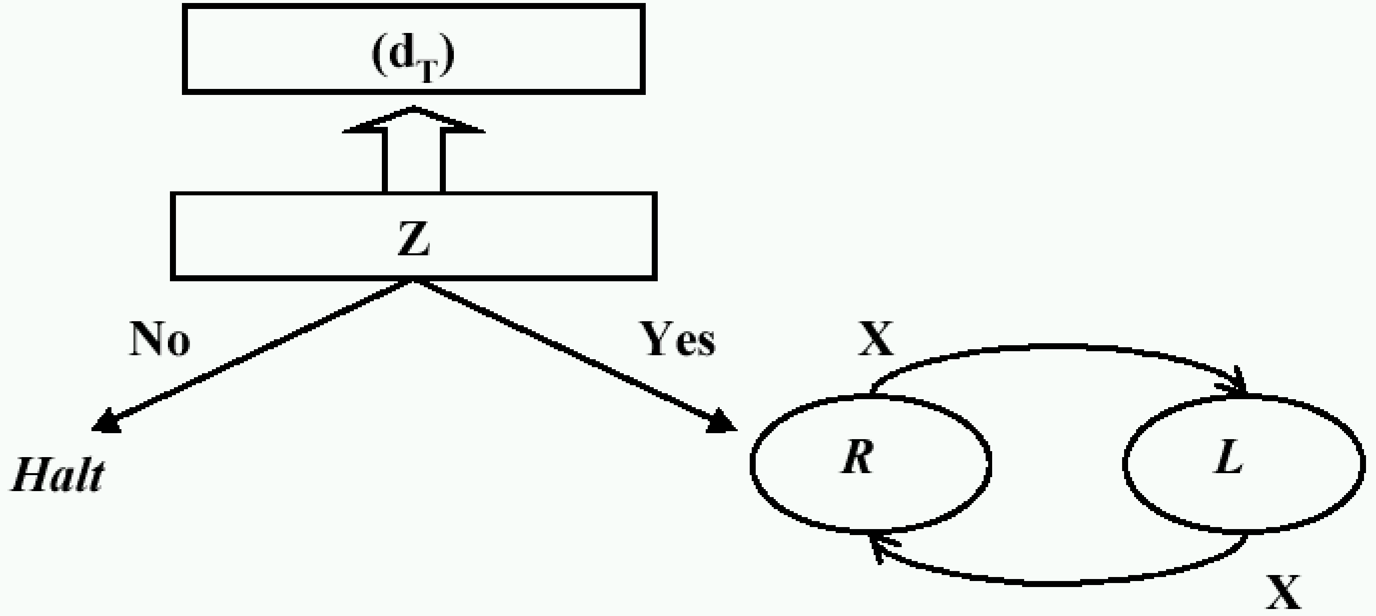


We create machine E

Proof (cont)

Step 2: Introduce a machine **Z**, that prevents **E** from halting if **E** takes the Yes route.

In other words, if E spits out "no", Z does halt, if it spits out yes, it does not halt, but enters an infinite loop.



We create machine Z

Proof (cont)

- In summary, we have:
  - **T(d_T) halts ==> Z(d_T) does not halt**
  - **T(d_T) does not halt ==> Z(d_T) halts**
- Step 3:
  - Let us write a description **d_Z** for **Z** and substitute Z for T in the foregoing argument:

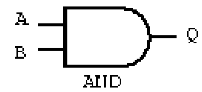
  - ***Z(d_Z) halts iff Z(d_Z) does not halt***
- This is a contradiction and therefore, the assumption that a machine **D** exists was wrong!

Question: *Is it possible to build a machine that will tell us whether a Turing machine T* ***with tape t*** *will halt?*
Answer: *No, this is not possible!*

….in addition, this proof illustrated typical proof techniques in undecidability theory…..

Circuits

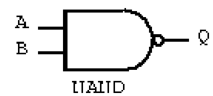
- Circuit : wires + gates
- Gates : function $\{0,1\}^k \rightarrow \{0,1\}^m$
- No loops
- Elementary circuits
  - AND, OR, NOT, NAND, NOR, XOR
  - Fanout
  - Crossover
  - Work (ancilla) bits



AND

$Q = A \cdot B$

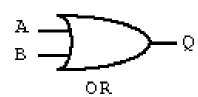
A	B	Q
0	0	0
0	1	0
1	0	0
1	1	1



NAND

$Q = \overline{A \cdot B}$

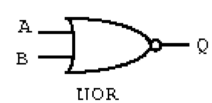
A	B	Q
0	0	1
0	1	1
1	0	1
1	1	0



OR

$Q = A + B$

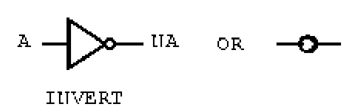
A	B	Q
0	0	0
0	1	1
1	0	1
1	1	1



NOR

$Q = \overline{A + B}$

A	B	Q
0	0	1
0	1	0
1	0	0
1	1	0



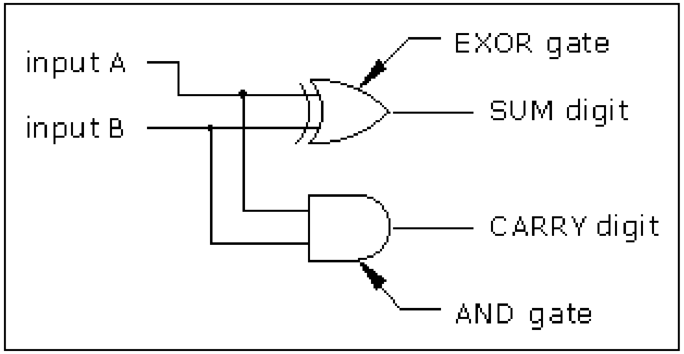
NOT

$Q = \overline{A}$

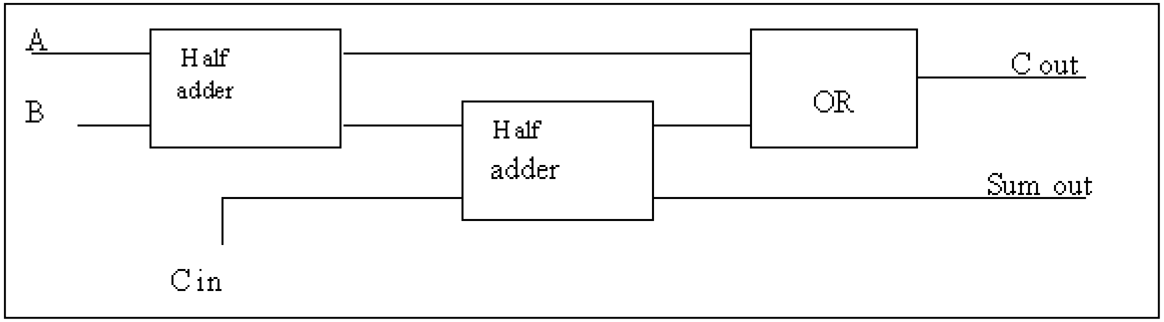
A	Q
0	1
1	0

Putting Circuits Together

- Here is a half-adder (half because doesn't take carry as in an input):



- Here is a full-adder then:



Universality of Circuit Model

- Any function $\{0,1\}^k \rightarrow \{0,1\}^m$ can be computed with:
  - Wires
  - Work bits (*ancilla bits*) prepared in some fixed state
  - Fanout operation
  - Crossover
  - AND, XOR gates (or just NAND)
- *How is crossover different from crossed wires?*
- *Why cannot we do crossover with XOR?*

Families Of Circuits

- A single function $\{0,1\}^k \rightarrow \{0,1\}^m$ is merely a 2^k row look-up table.
- Obviously, any such function is computable.
- Furthermore, it doesn't correspond to our notion of algorithm which can be defined for arbitrarily large numbers (e.g., $f(n)= n^2$)
- What do we do?
- We will introduce families of circuits…

Families Of Circuits

- Thus, define a uniform circuit family as:
  - a set $\{C_n\}$ of circuits
  - C_n handles inputs of size up to n
  - For all $m>n$ for all x $\mathbf{C_m(x)=C_n(x)}$
  - There is a Turing machine T_C such that $T_C(n)$ produces description of circuit C_n
- Then $\{C_n\}$ computes function $C()$ ***iff*** for all x $C(x)=C_{|x|}(x)$
- this means : **complex gates = subroutines**
- Uniform circuit families are equivalent to Turing machines
- Therefore they can **compute anything computable**
  - **For any circuit we can create TM**
  - **For any TM we can create a circuit**
- **It is also obvious from Shannon or Davio expansions that every function of any number of inputs can be build from C_n see page 133 in the textbook.**

Turing Machine Countability

- All Turing machines **and/or** circuits can be algorithmically enumerated
- It means that:
  - **1.** Every possible Turing machine/circuit can be assigned a unique integer ID number
  - **2.** There is a Turing machine/circuit that given index j produces the full description of Turing machine/circuit #j

Turing-Church Thesis

- Anything that can be computed mechanically/algorithmically can be computed on a Turing machine
- Corollary: anything that can be computed mechanically/algorithmically can be computed with a uniform family of circuits
- **Proof?**
- "computed mechanically/algorithmically" is too fuzzy to use in a proof…

Strong Turing-Church Thesis

- Any model of computation can be simulated on a probabilistic Turing machine with at most polynomial increase in the number of elementary operations required.

This thesis implies that attention may be restricted to the probabilistic Turing Machines.

Quantum Computers cast in doubt Strong Turing-Church thesis, by enabling the efficient solution of a problem which is believed to be intractable on all classical computers, including probabilistic Turing Machines.

Computational Complexity Classes

- *(Computational) Complexity* refers to some measure of the resources required to solve a problem. We will restrict attention to *decision problems*.
- Decision problems = Yes or no Answers.
- Decision problems can be treated as the problem of recognizing elements of a *language*.

Formal languages: a language L over alphabet Σ is a subset of the set Σ^* of all (finite) string of symbols from Σ. Example: if $\Sigma = \{0, 1\}$, then $L = \{0, 10, 100, 110, \ldots\}$ is a language over Σ.

Review: What is a language?

- Fix an alphabet, say $\Sigma = \{0,1\}$. The set Σ^* denotes all finite length strings over that alphabet.
- A *language* L is a subset $L \subseteq \Sigma^*$
- An algorithm solves the language recognition problem for L if it accepts any string $\sigma \in L$ and rejects any string $\sigma \notin L$

What is a language?

•**Decision problems:** **problems with yes or no answer.**
•**Example:** **Is a given number a prime? (Primality decision problem)**

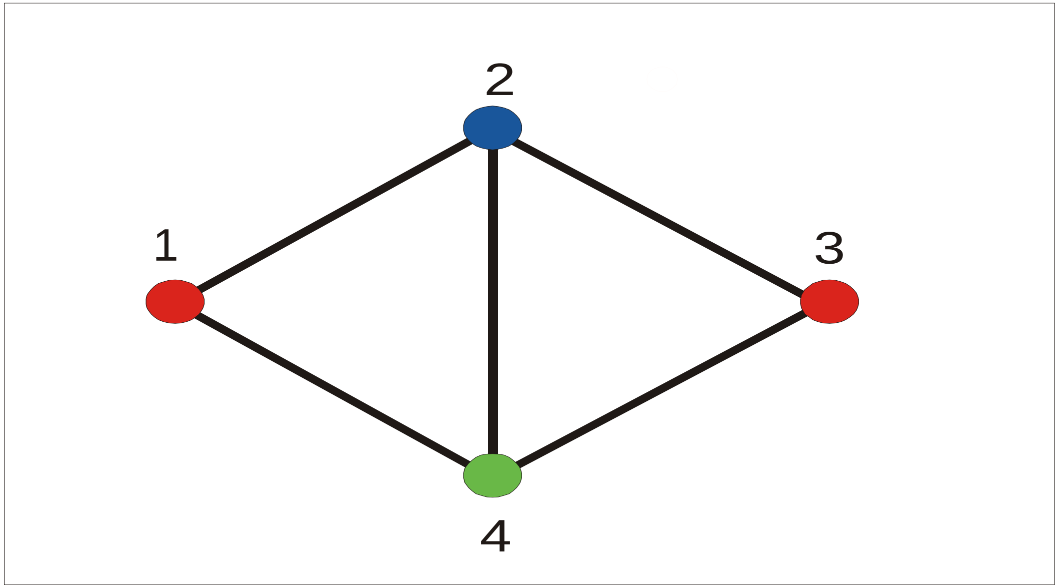
- E.g. $PRIME = \{10, 11, 010, 011, 101, 111, \ldots\}$

$$COMPOSITE = \{100,110,0100,0110,1000,...\}$$

- If we let strings x represent a graph, then we can define

$$3-COLOURABLE = \{x \mid x \text{ is properly } 3-\text{colourable}\}$$

What is a language?



- The string 101111 represents this graph by telling us which pairs of vertices {1,2},{1,3},{1,4},{2,3},{2,4},{3,4} are connected. This graph is 3-colourable so

$$101111 \in 3-\mathrm{COLOURABLE}$$

Computability / Decidability versus Languages

- Decidability problems are often described by **languages:**
  - the input are members of a larger set
  - the output is **Yes/No** on whether the input belongs to a given language (set) L

- Examples:
  - L={n|n is a prime number}
  - L={n|n is an index of Turing machine that halts on input 0}

decidable

undecidable

Why A Formalization is Necessary?

- Allows us to answer several questions:
  - What is a computational **problem**?
  - Is there an algorithm to solve it?
  - What are the minimal resources to solve a problem?
    - Resources: time, space, and energy.
    - Can we classify the problems according to the resource requirements needed to solve them?

…. Now we will be trying to explain the concept of decidability/undecidability…….

Single Instance Problems

- It is not interesting to pose a single instance problem
- **For example:** is the problem of answering "Can machines think?" computationally decidable?
- Sure – there exists a program that prints out "Yes" and there exists a program that prints out "No"
- By our definition on the previous slide the problem is decidable

...but this is not what we mean…....

Computable functions

- What does it mean if a function is computable?
- What device is to be used?
- **Does it matter?**

- Turing machines can compute anything computable – thereby formalizing the definition of computability

Decidability

- Decidability == computability of the corresponding decision mass problem
  - What is a computational problem?
    - Here : a mass problem with ‘yes’/’no’ answer
  - Is there an algorithm to solve it?
    - **Not always**

Mass Problems

- Consider this problem:

"Is X entailed by a set of axioms?"

- The answer is '***yes***' or '***no***' as X is just a single constant expression
- Therefore, there exists a program which takes X and outputs 'yes' or 'no' (it will just contain one print statement (e.g., print('yes')))
- We might not know how to write that program but it trivially exists

Mass Problems

- To make *computability/decidability definitions* meaningful we can use **mass problems**
- A mass problem consists of inputs and desired outputs (e.g., $n \rightarrow n^2$)
- Mass Yes/No problems ➔ mass decision problems
- We say that a mass problem is computable/decidable ***iff*** there exists an algorithm for finding the desired answer for **any valid input**

Another Example

- Mass problem :

"Given a Turing machine number m, can we algorithmically determine if T_m will halt on the empty input?"

- another formulation of the Halting Problem
- It is **Undecidable**

Two Examples of Mass Problems

- In class we show two examples of mass systems:
  - First Order Predicate Calculus
  - Markov algorithms (variant of Post correspondence problem).
- In the ***second problem*** you have an alphabet of characters and two strings over this alphabet, S1 and S2.
- You have also a set R or rewriting rules which are pairs of sequences of characters.
- The question is, can S2 be obtained from S1 by applying the rules from R.
- Because the rules create both longer and shorter strings, you cannot create a program that would in finite time derive S2 from S1 or tell that it is not possible.
- This is an example of an undecidable problem.
- Note that we are not asking for a solution to any particular problem of this type, but for the existence of a procedure to decide for any set of rules R and any two strings S1 and S2 - thus a mass problem.
- **See more on these two examples on the WWW page of the class.!!**

Computable Function for a Mass Problem?

- Remember: The problem of proving a given FOPC (First Order Predicate Calculus) statement is **undecidable**
- Thus, need a mass problem:

"Is there a computable function f(X) such that:

1) f(X)='yes' iff X is a FOPC statement entailed by a given set of axioms

and

2) f(x)='no' iff X isn't." ??

- Undecidable – no such computable function exists

Decision Problems

- The answers are:

  - What is a computational problem?
    - Here : a mass problem with ‘yes’/’no’ answer
    - Example: “Function f(n) such that f(n)=yes iff n is prime and f(n)=no iff n is not prime”

Computational complexity

- Computational complexity is the **study of the time and space** resources required to solve computational problems.
- Task: prove lower bounds on the resources required by the best possible algorithm for solving a problem.
- Suppose that the problem is specified by giving n bits as an input.
  - Chief distinction: problems which can be solved using the resources which grow polynomial in n and problems which grow faster than any polynomial in n.
- The problem is regarded as easy, tractable or feasible if an algorithm for solving the problem using polynomial resources exists, and as hard, intractable or infeasible if the best possible algorithm requires exponential resources.

Computability versus Decidability

- A problem is computationally solvable (or computable) if there exists a program that computes the answer

- If the answer is of the Yes/No type then the problem is called a decision problem

- If such a problem is computable we say it is decidable

Examples of doable tasks

- Fortunately, some tasks are more doable
- Examples:

1. *"For any given 3 numbers a,b,c return 'yes' if $a=bc$ and 'no' otherwise"*

2. *"For any given number n return its prime factors"*

- Both are computable
- But the complexity is different
- So need finer distinctions

Complexity

- Now we are more detailed with answers:
  - What is a computational problem?
    - Here : a mass problem with ‘yes’/’no’ answer
  - Is there an algorithm to solve it?
    - Not always
  - What are **the resources** to solve a problem?
    - Coarse division : tractable / intractable
    - Finer division : asymptotic notation

Asymptotic Notation

- Big O : $f=O(g)$ ***iff*** there exists a constant c that starting from some x_0 holds $f(x) < cg(x)$ (i.e., g upper-bounds f)
- Example: $\text{sum}_{i=1..n}\ i = O(n^2)$
- Good for worst-case performance analysis
- Example: linear search is $O(n)$

Asymptotic Notation

- Lower bound
- Big Omega : $f=\Omega(g)$ ***iff*** there exists a constant $c\neq 0$ that starting from some x_0 holds $f(x) > c\ g(x)$ (i.e., g lower-bounds f)
- Sometimes used for the best case analysis
- Example: any binary-comparison based sorting is $\Omega(n \log n)$

Coarse Division : Tractable/Intractable

- Often we want to make a statement if an algorithm is tractable/feasible or intractable/infeasible

- The crude formalization is this:

If the worst case running time is polynomial (i.e., $O(n^k)$ where k is a constant) then the algorithm is tractable in running time

- Here n is the input size in a reasonable (e.g., binary) representation
- The running time measured on a deterministic Turing machine

Slightly finer division

- Class P – time to ***solve***: O(poly(|input|))

- Class NP – time to ***verify*** : O(poly(|input|))
- Class NP-complete -- any other NP problem is ***reducible to it***
- Class NPI – NP ***but not*** NP-complete

- Class PSPACE -- ***space*** ***to solve***: O(poly(|input|))
- Class EXP – ***space to solve*** : $O(2^{poly(|input|)})$

Class P

- Thus can define:

P (polynomial time) is the class of languages that can be decided by a deterministic Turing Machine running in time **$O(n^k)$.**

… more precisely….

- The class P consists of all languages L for which there exists a classical algorithm A running in worst-case polynomial time such that for any input $x \in \Sigma^*$ the algorithm A on input x, A(x), accepts if and only if $x \in L$

Class P

- Examples:
  - Search: $n < n^1$
  - Sorting: $n \log n < n^2$
  - Etc.

- Counter examples:
  - Sure, take a number n, idle for 2^n time ticks, output ‘yes’. This algorithm is exponential but the function it represents is $O(1)$

The complexity class P in terms of languages

- A problem is to be said to be solvable in polynomial time if it is in **TIME(n^k)** for some finite k.
- The collection of all languages which are in **TIME(n^k)**, for some k, is denoted P.
- A complexity class is defined to be a collection of languages.
- Unfortunately, proving that any given problem can't be solved in polynomial time seems to be **very difficult**!
- ***Example:*** the factoring decision problem is believed not to be in P.
  - (Given a composite integer m and l <m, does m have a non-trivial factor less than l?)

Number factorization is tough

Class NP

- **Some problems appear harder**
- **Example: "Is a given number composite (i.e., not prime)?". No polynomial algorithm is known.**
- NP = Non-Deterministic Polynomial Time is the class of languages that can be verified by a deterministic TM running $O(n^k)$ time.
- Example: $L = \{l \in \boldsymbol{N} \mid l \text{ is not prime}\} \in NP$.
- Problems in NP: yes answer can be easily verified with the aid of an appropriate **witness**.

Factoring Decision Problem

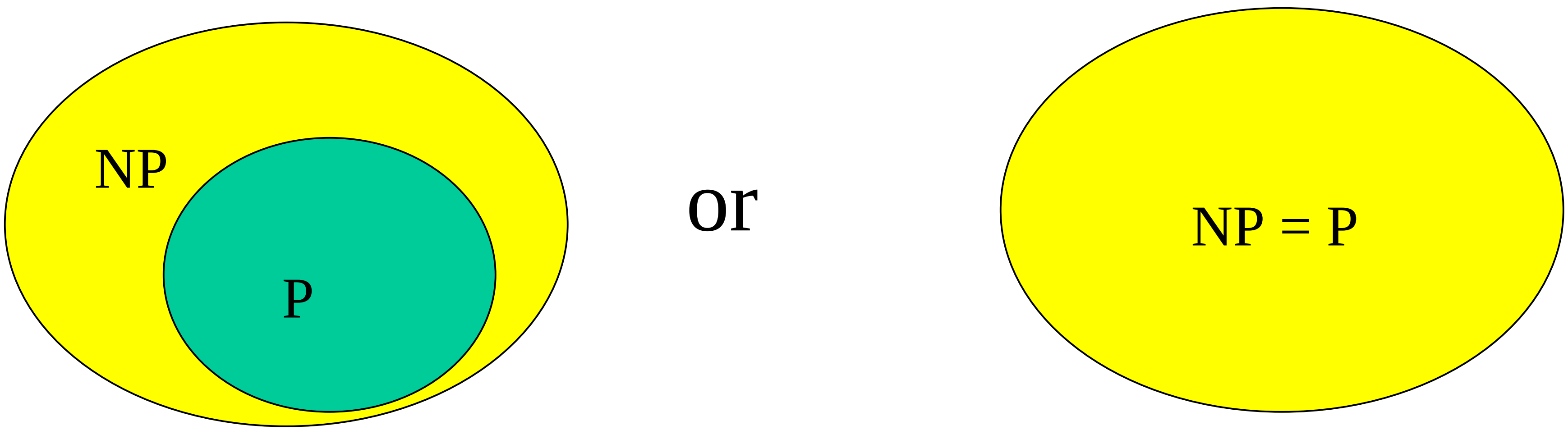
Given a composite integer m and *l*<m, does m have a non-trivial factor less than *l*?

The complexity class NP

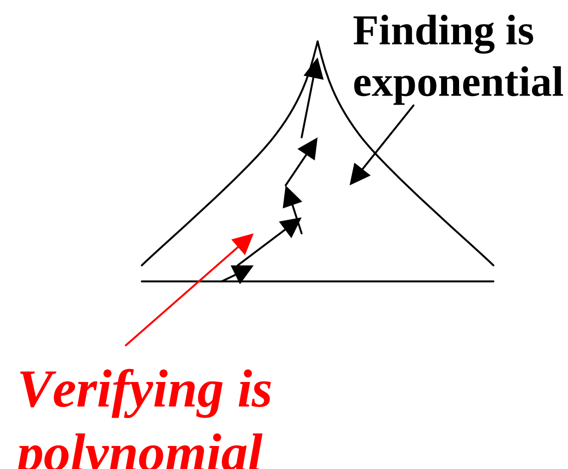
- A language L is in NP if the is a Turing machine M with the following properties:
  - **(1)** If $x \in L$ then there exists a **witness string w** such as that M halts in the state q_Y after a time polynomial in |x| when that machine started in the state x-blank-w.
  - **(2)** If $x \notin L$ then for **all strings w** which attempt to play the role of a witness, the machine halts in state q_N after a time polynomial in |x| when M is started in the state x-blank-w.
- There is an apparent asymmetry in the NP definition: *it is easy to decide whether a possible witness to $x \in L$ is truly a witness.*

How are NP and P related?

- P is a <u>subset of</u> NP.
- It is not known whether or not there are problems in NP which are not in P.



Discuss CSAT, versus SAT, versus 3SAT, versus 2SAT - 2SAT is in P!, all other in NP.



A Corollary about witnesses

- Given an input x of size $|x|=n$ and an appropriate witness w there must be a polynomial time <u>algorithm</u> to check if x belongs to L
- This means that $|w|=O(poly(|x|))$

- **Why?**
- Otherwise, the Turing Machine won't be even able to read in w

Class Co-NP

- What about "Is n prime?"
- Can easily check if n is <u>not</u> a prime if given a witness (e.g., a factor of n)

- Define:

coNP is the class of languages where we require a witness for every negative (i.e., $\ell \notin L$) instance. Clearly, $\forall L[L \in \mathbf{NP} \iff \overline{L} \in \mathbf{coNP}]$. Example: $\overline{L} = \{\ell \in \mathbb{N} | \ell \text{ is prime}\} \in \mathbf{coNP}$.

… witness for every negative instance…..

Class NP-Complete

- Some **NP-problem** are **especially hard** insomuch as any other NP problem can be reduced to any of them
- **Reduction :** if I have a NP decision problem L (i.e., I am asking a question "Is x in L?") and an NP-complete problem M then for any x it takes polynomial time to produce y such that y is in M ***iff*** x is in L
- In other words, the time complexity of L is O(poly(t)) where t is the time complexity of M

Class NP-Complete

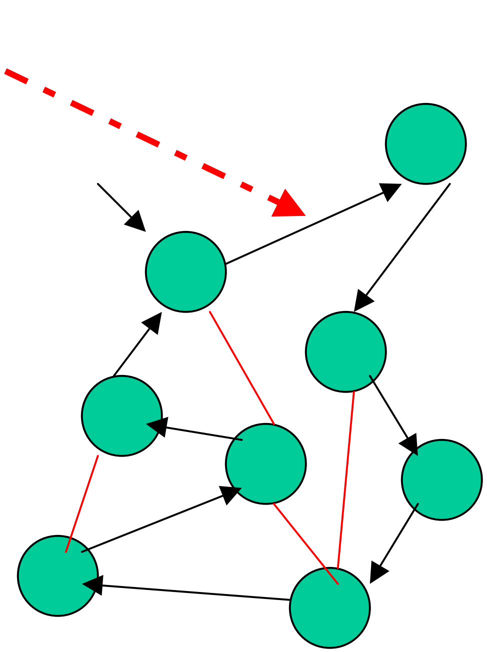
- Formally:

NP -complete is the class of languages from **NP** such that for any language L from **NP** L can be polynomially reduced to that **NP** -complete language;

- Examples:
  - **CSAT** : given a Boolean circuit of AND and NOT gates, is there an assignment of its inputs such that the entire circuit produces 1 (true)?

Problem Example: Hamiltonian Cycle or HC

- Hamiltonian cycle is an ordering of all graph vertices such that no vertices are repeated except the starting vertex.
- The cycle has to have the edges present in the graph.

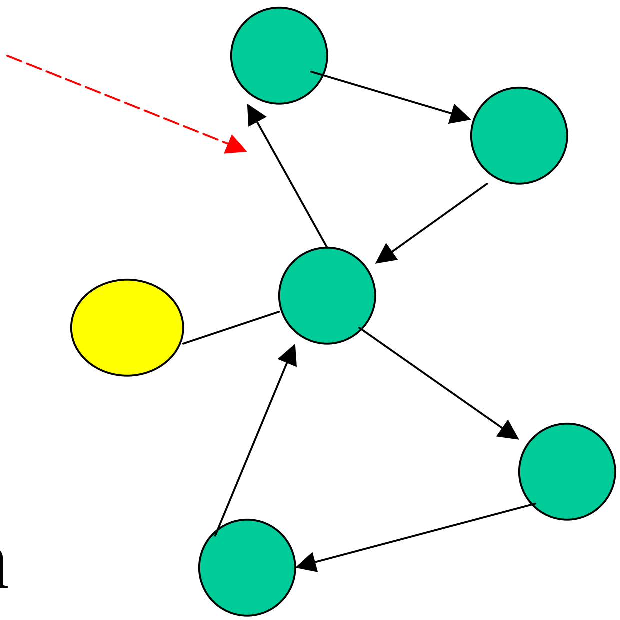


- **Decision-problem :** does a given graph have a Hamiltonian cycle?

…graph of green nodes has Euler cycle, with yellow node - not…..

Problem Example: Euler Cycle or EC

- Euler cycle is an <u>ordering of all edges</u> of a graph such that:
  - every edge is visited **<u>exactly once</u>**
  - any two consecutive edges in the sequence share a vertex
  - the sequence forms a cycle



- **Decision-problem :** does a given graph have an Euler cycle?

<u>Which of these problems is easy, which not?</u>

…graph of green nodes has Euler cycle, with yellow node - not…..

HC vs. EC

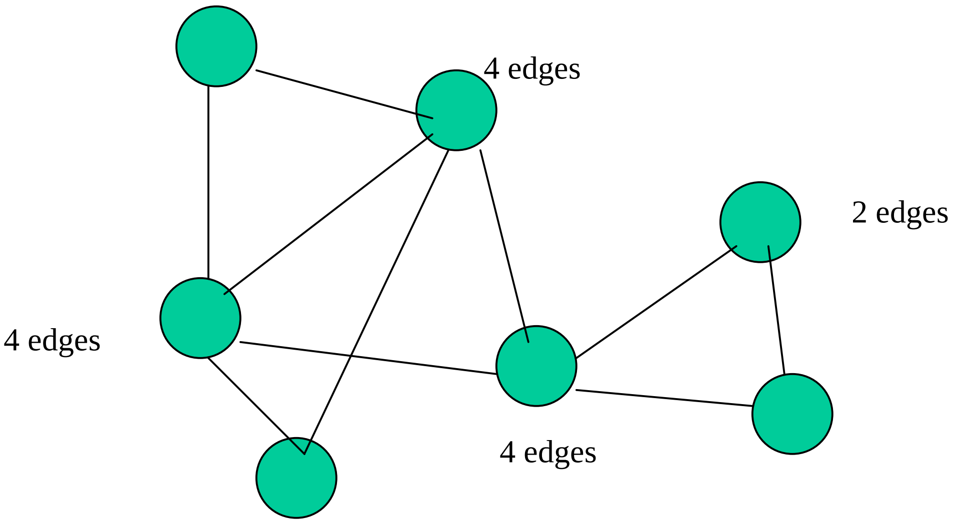
Surprize!!!

- Hamiltonian cycle is NP-complete

- Euler Cycle is in P (can be solved in $O(|\text{input}|^3)$)

Euler is easy!

- **Euler Theorem:**
  - A connected graph contains an Euler cycle if and only if every vertex has an even number of edges incident upon it.



Class NPI

- How about problems that are in NP but not NP-complete?
- They would belong to NPI (NP Intermediate)
- ***Do they exist?***
- Unknown but ***suspected*** that:
  - Factoring is in NPI
  - Graph isomorphism is in NPI

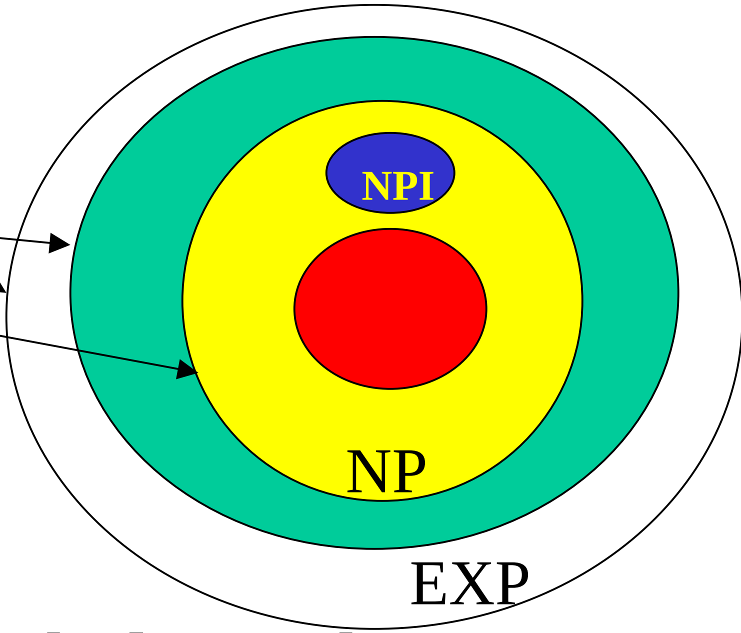
May be this class is empty?

Classes PSPACE & EXP

- **PSPACE**: Problems that can be decided in ***space*** $O(\mathrm{poly}(|\mathrm{input}|))$
- **EXP**: Problems that can be decided in ***space*** $O(2^{\mathrm{poly}(|\mathrm{input}|)})$

Class Inclusion

- What do we know?
  - $P \subseteq NP \subseteq PSPACE \subseteq EXP$
  - NP-complete $\subseteq$ NP
  - NPI $\subseteq$ NP
  - $P \subset EXP$



- What don’t we know but really believe that it is true?
  - $P \subset NP$?
  - NPI $\neq \emptyset$?
  - $P \subset SPACE$?

A complexity class: BPP

- The class BPP (*bounded-error probabilistic polynomial time*) consists of all languages L for which there exists a randomized classical algorithm A running with worst-case expected polynomial time such that for any input $x \in \Sigma^*$

$$\text{If } x \in L \Rightarrow \Pr[A(x) \text{ accepts}] \geq \frac{2}{3}$$

$$\text{If } x \notin L \Rightarrow \Pr[A(x) \text{ accepts}] \leq \frac{1}{3}$$

- nota bene we are not averaging over x

Chernoff bound and BPP

- Is $\frac{2}{3}$ special? No. $\frac{1}{2}+\delta, \delta > 0$ suffices.
- We can repeat the algorithm A n times and take the majority answer. We now get the correct answer with probability at least $1-\varepsilon^n$ for some $\varepsilon, 0 < \varepsilon < 1$ (see Box 3.4 in the text)

BPP $\approx$ Efficient??

- We view decision problems corresponding to recognizing languages in BPP as tractable
- We view problems without such worst-case polynomial time solutions as intractable.

Polynomial time $\approx$ Efficient??

- "It should not come as a surprise that our choice of polynomial algorithms as the mathematical concept that is supposed to capture the informal notion of *'practically efficient computation'* is open to criticism from all sides. [...]

Polynomial time ≈ Efficient??

Ultimately, our argument for our choice must be this: **Adopting polynomial worst-case performance as our criterion of efficiency results in an elegant and useful theory that *says something meaningful about practical computation*, and would be impossible without this simplification**" – Christos Papadimitriou

So how it relates to quantum?

- Well, we know that:
  - Polynomial **quantum** algorithms are in PSPACE
- It's believed:
  - Polynomial **quantum** algorithms can do MORE than polynomial classical algorithms
  - ***Specifically:*** they can do NPI but NOT all NP (i.e., not NP-complete)

Feynman's question

- The second track to quantum computation.
  - R.P. Feynman, 1982

    Simulating physics with computers,

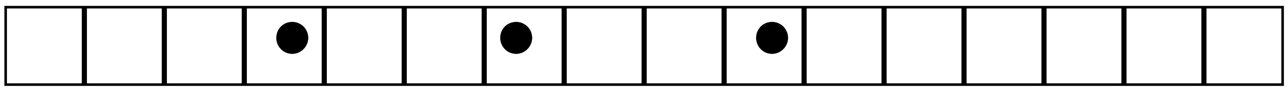
    Int. J. Theor. Phys. **21**, 467 (1982).

- Can a *quantum* system be **simulated** ***exactly*** by a universal computer ?

NO !

Classical simulation: transport problem

- Simulate Boltzmann equation.



- R particles on a 1-dim lattice of N sites.
  - note, for fields R=*O* (N)
- How does the calculation scale with N,R ?

$$\text{size of input} \approx N^{2R}$$

Classical probabilistic simulation.

- Use random numbers to *simulate* coarse grained dynamics.
- The statistics of random numbers is *classical*.
- Cannot simulate a large quantum process.

The Feynman processor

- A physical computer operating by quantum rules.
  - could it compute more *efficiently* than a classical computer ?

Quantum Turing Machines

- Church-Turing Thesis:
  - Every “function which would naturally be regarded as computable” can be computed by the universal Turing machine.
- Quantum Turing Machine
  - can compute partial recursive functions
  - can simulate any quantum computer with arbitrary precision

Deutsch and quantum parallelism

- D. Deutsch, 1985

  Quantum theory, the Church-Turing principle and the universal quantum computer.

  Proc. Roy. Soc. A**400**, 97, (1985).

- Feynman-Deutsch principle:

  (Church-Turing principle)

  'Every finitely realisable physical system can be perfectly simulated by a universal model computing machine operating by finite means"

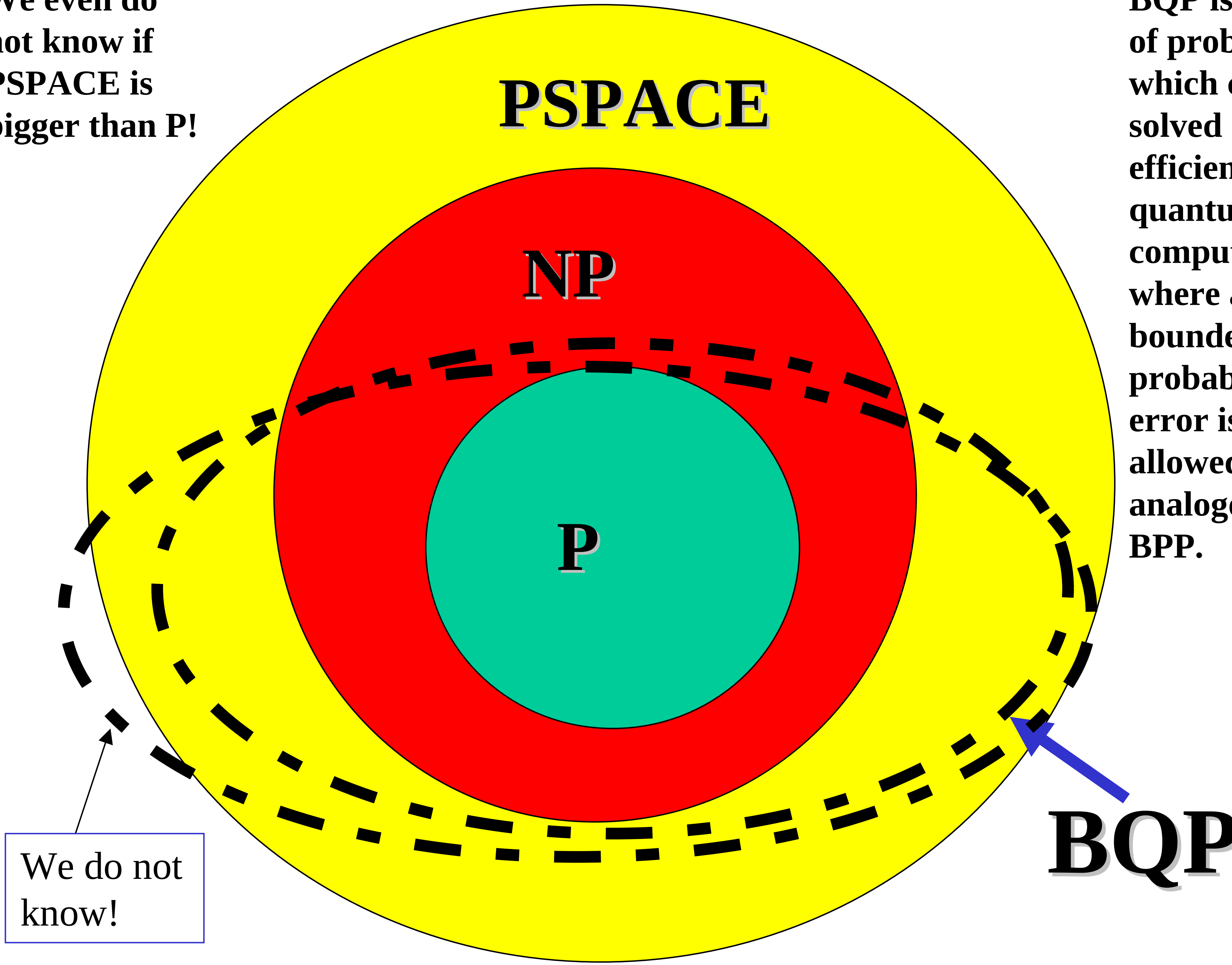
Deutsch processor

- Computational basis:
  - Direct product Hilbert space of N two-level systems: $|S_N\rangle \otimes |S_{N-1}\rangle \otimes \ldots |S_1\rangle; \quad S_i \in \{1,0\}$
- Quantum Turing machines:
  - remain in computational basis state at end of each step.
- Quantum computer
  - arbitrary superpositions of computational basis...explore all 2^N dimensions !

Computational Complexity Classes - role of quantum

- Finding non-trivial lower bounds on the worst-case complexity of computational problems has proved very difficult
- We hope that this more general framework of quantum computation will help us find non-trivial lower bounds and some new relationships between complexity classes
  - (*like complex numbers help us understand real numbers*)

We even do not know if PSPACE is bigger than P!



BQP is a class of problems which can solved efficiently on a quantum computer where a bounded probability of error is allowed - analogous to BPP.

Exam Problems.

- The material in this lecture is advanced. Do not worry if you have some troubles. You have however understand the following topics:
- Definition of P problems. Definition of NP problems.
- Formulation of P=NP controversy and its practical and philosophical meaning.
- Classes of complexity and what problems belong to it.
- Physical versus mathematical unsolvability/undecidability.
- Meaning of Goedel and Turing results. Explain in your own words and illustrate.
- The concept of mass problems. Why different from single instance problems? Examples.
- Examples of problems and their complexity classes.
- Relations between class NP and undecidable problems. How can the quantum computer help, can it?
- Give examples of halting problem, also other examples than those from lecture or book.
- Explain why predicate calculus and Markov algorithm (Post equivalence Problem) are undecidable?
- Why are problems of Artificial Intelligence and undecidability related?

Exam Problems.

- Discuss complexity of Factoring and other similar problems.
- Examples of simple Turing Machines.
- Idea of Universal Turing Machine
- Circuits versus Turing Machines - why equivalent?
- Formulate a Turing Machine that has a subroutine NAND that calculates function OR of two arguments. NAND is a NAND of two inputs.
- Link the language concept to undecidability and complexity
- Class BPP
- Discuss quantum complexity issues.