INTRODUCTION TO THE THEORY OF NP-COMPLETENESS

CSI4105: LECTURES 1-5

Lucia Moura

University of Ottawa

Fall 2003

These notes/slides are intended as an introduction to the theory of NP-completeness, as a supplementary material to the first sections in Chapter 34 (NP-completeness) of the textbook:

Cormen, Leiserson and Rivest, Introduction to Algorithms, 2nd ed, 2001.

Things that you will find here but not in this textbook include definitions of basic models of computation (Turing machines and the RAM model) and equivalence between models.
These concepts are important in order to have a formal definition of "algorithms" and of their running times (rather than intuive notions only).

These notes will take you from Letures 1 to 5, and include the material in sections 34.1 and 34.2. After that, we will closely follow the textbook and no notes will be provided.

Other references for these notes:

Garey and Johnson, Computers and Intractability: a guide to the theory of NP-completeness, 1979.

Sipser, Introduction to the Theory of Computation, 1996.

Papadimitriou, Computational Complexity, 1994.

A GENERAL INTRODUCTION TO NP-COMPLETENESS

A practical application

Suppose your boss asks you to write an efficient algorithm to solve an extremely important problem for your company.

After hours of hair-splitting, you can only come up with some "brute-force" approach that, since you took CSI3501, you are able to analyse and conclude that it takes exponential time!

You may find yourself in the following embarrassing situation:

/—— PICTURE ——/
(from Garey and Johnson page2)

"I can't find an efficient algorithm, I guess I'm just too dumb."

You wish you could say to your boss:

/—– PICTURE —-/
(from Garey and Johnson page2)

"I can't find an efficient algorithm, because no such algorithm is possible!"

For most problems, it is very hard to prove their intractability, because most practical problems belong to a class of well-studied problems called NP.
The "hardest" problems in NP are the **NP-complete problems**: if you prove that one NP-complete problem can be solved by a polynomial-time algorithm, then it follows that all the problems in NP can be solved by a polynomial-time algorithm. Conversely, if you show that one particular problem in NP is intractable, then all NP-complete problems would be intractable.

- NP-complete problems seem intractable.
- Nobody has been able to prove that NP-complete problems are intractable.

By taking this course and mastering the basics of the theory of NP-completeness, you may be able to prove that the problem given by your boss is NP-complete.

In this case, you can say to your boss:

/—– PICTURE —-/
(from Garey and Johnson page3)

"I can't find an efficient algorithm, but neither can all these famous people."

or alternatively:

"If I were able to design an efficient algorithm for this problem, I wouldn't be working for you! I would have claimed a prize of $1 million from the Clay Mathematics Institute."

After the second argument, your boss will probably give up on the search for an efficient algorithm for the problem.

But the need for a solution does not disappear like that...

Indeed, after proving that your problem is NP-complete, you will probably have learned a lot about it, and now you can change your strategy:

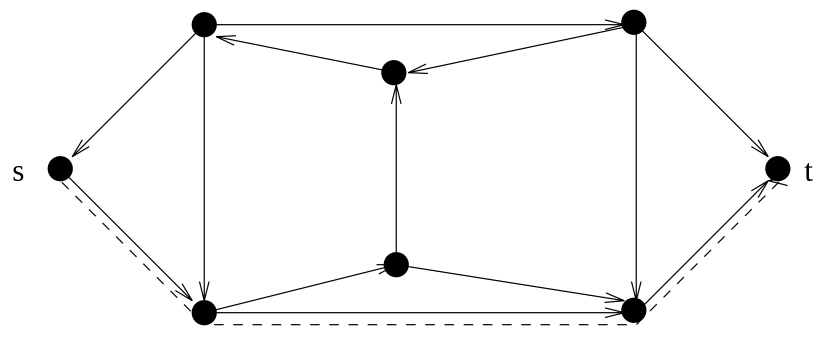
1) Forget about finding a polynomial-time algorithm for the general problem.

2) Try one of the following main approaches:

- Search for an efficient algorithm for a different related problem (relax some constraints, solve various special cases of the general problem or accept approximations in the place of optimal solutions)
- Use the best algorithm you can design for the general problem, even if it may take time that is exponential on the input size. (This may be enough to solve the instances of the problem that your boss needs, but performance is expected to deteriorate rapidly as the instance input size grows.)

Two Examples

Let G be a directed graph.
PATH problem: Is there a path in G from s to t ?



I claim that a polynomial-time algorithm exists for solving PATH.
Exercise: Give a polynomial time algorithm for *deciding* whether there exists a path from s to t.

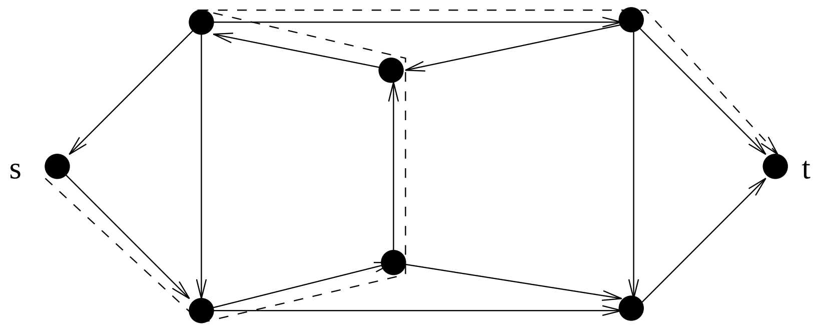
One possible algorithm: Breadth-first search for solving PATH:

```
1. mark node s
2. repeat until no more additional nodes are marked:
     for each edge (a,b) of G do
        if (node a is marked and node b is unmarked)
              then mark node b
3. if t is marked then return "yes"
                  else return "no".
```

Thus, PATH is in the class **P**, the class of problems that can be solved by a polynomial-time algorithm.

Consider a slightly different problem:

HAMPATH problem: Is there a **hamiltonian** path in G from s to t ?
(Def: A path is called *hamiltonian* if it goes through every node exactly once.)



Exercise: Give a polynomial-time algorithm to solve HAMPATH.

(If you find a solution to this problem, talk to me.)

Exercise: Give any algorithm to solve HAMPATH.

One possible algorithm: Brute-force algorithm for HAMPATH:

`IDEA:` Let V be the vertices in G and let $n = |V|$. Check all possible lists of n vertices without repetitions (all permutations of V) to see if any of them forms a hamiltonian path.

```
1. for each permutation
   p=(v(i(1)),v(i(2)),...,v(i(n))) of V do:
        if (p forms a hamiltonian path)
            then return "yes"
2. return "no"
```

What has to be checked in order to see that `p` forms a hamiltonian path?

Why does this algorithm run in exponential time ?

Nobody knows of an algorithm that solves HAMPATH in polynomial time.

Can we check a solution in polynomial time?

Suppose someone says that our graph has a hamiltonian path, and provides us with a certificate: a hamiltonian path.

Can an algorithm *verify* this answer in polynomial time?

The answer is yes. *Exercise:* Give such an algorithm.

NP is defined as the class of problems that can be *verified* in polynomial time, in order words, the problems for which there exists a certificate that can be checked by a polynomial-time algorithm.

Therefore, HAMPATH is in the class **NP**.

The name **NP** stands for "Nondeterministic Polynomial-time" (this will be more clear later).
It does NOT stand for "NonPolynomial" !!!!!!

Indeed, later on, you will be able to easily prove that $\mathbf{P} \subseteq \mathbf{NP}$.
(Intuitively, it seems reasonable that if a problem can be solved by an algorithm in polynomial time, then a certificate for the problem can be checked in polynomial time.)

NP-completeness

Let us concentrate on *decision* problems, that is, problems whose solution is “yes” or “no”, like the versions of PATH and HAMPATH seen before.

In summary:
P is the class of decision problems that can be **decided** by a polynomial-time algorithm.
NP is the class of decision problems such that there exists a certificate for the yes-answer that can be **verified** by a polynomial-time algorithm.

Also,

$$\mathbf{P} \subseteq \mathbf{NP}$$

One of the most important unsolved problems in computer science is the question of whether $\mathbf{P} = \mathbf{NP}$ or $\mathbf{P} \neq \mathbf{NP}$.

See: `http://www.claymath.org/prizeproblems/index.htm` for details on how to get a $1 million prize for the solution of this question.

How one would try to solve this question ?

To show that $\mathbf{P} \neq \mathbf{NP}$:

Show that a problem X is in the class **NP** but is not in the class **P**.

Exercise: What would have to be shown about X ?

To show that $\mathbf{P} = \mathbf{NP}$:

Show that every every problem in **NP** can be solved in polynomial time, so that $\mathbf{NP} \subseteq \mathbf{P}$.

Given the theory of NP-completeness, we don't need to work that hard. We could show that $\mathbf{P} = \mathbf{NP}$ by finding a polynomial-time algorithm for a single problem such as HAMPATH. This is not obvious at all!!!

Before understanding NP-completeness, we need to understand the concept of polynomial-time reducibility among problems.

Intuitively, a problem Q_1 is *polynomial-time reducible* to a problem Q_2 if any instance of Q_1 can be "easily rephrased" as an instance of Q_2.
We write:

$$Q_1 \leq_P Q_2$$

A polynomial-time algorithm that solves Q_2 can be used as a subroutine in an algorithm that can thus solve Q_1 in polynomial-time.

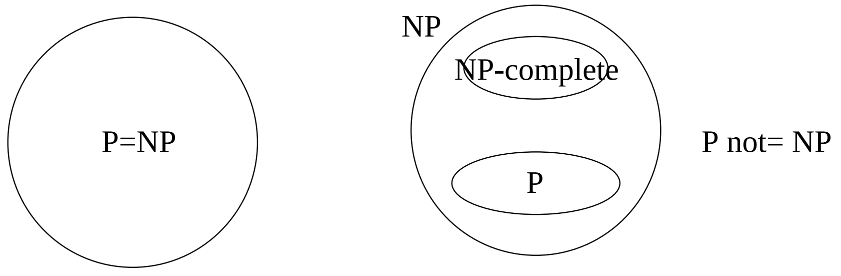
A problem Q is said to be **NP-complete** if:

1. Q is in the class NP.
2. $Q' \leq_P Q$, for every Q' in the class NP
   (every problem in NP is polynomial-time reducible to Q)

It is known that the problem HAMPATH is NP-complete.

The following theorem will be proven later in the course:

THEOREM. If some NP-complete problem is polynomial-time solvable, then $\mathbf{P} = \mathbf{NP}$. If some problem in NP is not polynomial-time solvable, then $\mathbf{P} \neq \mathbf{NP}$ and all NP-complete problems are not polynomial-time solvable.



Formalizing NP-completeness

In order to be able to use the NP-completeness machinery and to understand without ambiguity the concepts discussed before, we need precise definitions of:

- Problem
- Algorithm

It will take us several classes in order to define the required concepts appropriately and to formally present the theory of NP-completeness.

After that, you will learn how to show that several problems are NP-complete.

The second part of the course will be devoted to two main approaches for dealing with NP-complete problems:

- approximation algorithms, and
- specific exponential-time algorithms (backtracking and branch-and-bound).

Homework: Try to find a precise and general definition of "algorithm".

DEFINING PROBLEMS AND ALGORITHMS

Abstract decision problems

An **abstract problem** Q is a binary **relation** on a set I of problem instances and a set S of problems solutions.
(Recall: each element in the relation is an ordered pair (i, s) with $i \in I$ and $s \in S$.)

The SHORTEST-PATH problem statement is: "Given an undirected graph $G = (V, E)$ and two vertices $u, v \in V$, find the shortest path between u and v."

Each instance is a triple $< G, u, v >$ and the corresponding solution is the shortest path $(v_1, v_2, \ldots, v_n)$.

A **decision problem** is a problem with yes/no solutions. An **abstract decision problem** is a **function** that maps the instance set to the solution set $\{0, 1\}$.

The decision problem version of SHORTEST-PATH can be written as follows.
The PATH problem statement is: "Given an undirected graph $G = (V, E)$, two vertices $u, v \in V$, and a non-negative integer k, is there a path between u and v with length **at most** k?"

Note that the problem PATH *here is different from the one given last class; last class the graph was directed and the decision problem was different.*

The abstract decision problem is represented by the function:
$\textsc{Path}$: $I \rightarrow \{0,1\}$
If $i =< G, u, v, k > \in I$ then

$$\textsc{Path}(i) = \begin{cases} 1, & \text{if there exists a path between } u \text{ and } v \\ & \quad \text{with length at most } k \\ 0, & \text{otherwise} \end{cases}$$

An **optimization problem** is a problem in which something has to be maximized or minimized.
Every optimization problem can be transformed into a decision problem.

minimization problem	Find the shortest path...	Is there a path of length **at most** k?
maximization problem	Find the longest path...	Is there a path of length **at least** k?

The theory of NP-completeness is restricted to decision problems.
This restriction is not so serious:
If an optimization problem can be solved efficiently, then the corresponding decision problem can be solved efficiently.
Why? How $\textsc{Path}$ can be solved using the solution for $\textsc{Shortest-Path}$?

The same statement can be re-written as follows:*If a decision problem is hard then the optimization problem is hard.*

Concrete decision problems

In order to solve a problem by computer, instances must be represented in a way the computer understands.

An **encoding** of a set I of abstract objects is a mapping e from I to a set of strings over an alphabet Σ (Σ has to have at least 2 symbols).
Example: For $I = \{0, 1, 2, 3, \ldots\}$ and $\Sigma = \{0, 1\}$ we can use the standard **binary encoding** $e : I \rightarrow \Sigma^*$, given by:
$e(0) = 0$, $e(1) = 1$, $e(2) = 10$, $e(3) = 11$, $e(4) = 100$, etc.

A problem whose instance set is the set of strings over Σ is a **concrete problem**. We can use encodings to map an abstract problem to a concrete problem (strings that have no corresponding instances can be mapped to the solution 0)

For an instance $i \in I$, the **length** of i, denoted by $|i|$, is the number of symbols in $e(i)$. Using the binary encoding above, $|0| = 1$, $|1| = 1$, $|2| = 2$, $|3| = 2$, $|4| = 3$, etc.

Later, we are going to measure running time as a function of the input length. But the length of an instance of the concrete problem depends heavily on the encoding used. Some encodings are polynomially related, so the polynomial-time solvability for problems using one encoding extends to the other. There are some possible "expensive" encodings, but we will rule them out.

Example: Encodings of $k \in \{0, 1, 2, \ldots\}$

encoding name	example	length of k using given encoding
unary encoding	$e_1(7) = 1111111$	$\lvert k\rvert_{e_1} = k$
binary encoding (base 2)	$e_2(7) = 111$	$\lvert k\rvert_{e_2} = \lfloor \log_2 k \rfloor + 1$
ternary encoding (base 3)	$e_3(7) = 21$	$\lvert k\rvert_{e_3} = \lfloor \log_3 k \rfloor + 1$
base b encoding (base b)	e_b	$\lvert k\rvert_{e_b} = \lfloor \log_b k \rfloor + 1$

Except for the unary encoding, all the other encodings for natural numbers have lengths that are polynomially related, since

$$\log_{b_1} k = \frac{\log_{b_2} k}{\log_{b_2} b_1} = c \cdot \log_{b_2} k$$

with $c = 1/(\log_{b_2} b_1)$ constant.

The unary encoding has length exponential on the size of any of the other encodings, since $|k|_{e_1} = k = b^{\log_b k} \approx b^{|k|_{e_b}}$, for any $b \geq 2$.

If we assume that we only use “reasonable” encodings, and not expensive ones (like the unary encoding for natural numbers), then the length of an instance does not depend much on the encoding.

Usually we will not specify the individual encoding, and you may think of the binary encoding as the standard one. But keep in mind that results for the binary encoding extends to most natural encodings (like the decimal, hexadecimal, etc.), except for expensive encodings (like the unary one).

So, $|k| \in O(\log k)$ for all reasonable encodings.

Formal-language framework

The formal-language framework allows us to define algorithms for concrete decision problems as "machines" that operate on languages.

Example:
Consider the concrete problem corresponding to the problem of deciding whether a natural number is prime. Using binary encoding, $\Sigma = \{0,1\}$, this concrete problem is a function:
$\textsc{Prime}$: $\{0,1\}^* \rightarrow \{0,1\}$ with
$\textsc{Prime}(10) = \textsc{Prime}(11) = \textsc{Prime}(101) = \textsc{Prime}(111) = \textsc{Prime}(1011) = \ldots = 1$,
$\textsc{Prime}(0) = \textsc{Prime}(1) = \textsc{Prime}(100) = \textsc{Prime}(110) = \textsc{Prime}(1000) = \textsc{Prime}(1001) = \ldots = 0$

We can associate with $\textsc{Prime}$ a language $L_{\textsc{Prime}}$ corresponding to all strings s over $\{0,1\}$ with $\textsc{Prime}(s) = 1$:

$$L_{\textsc{Prime}} = \{10, 11, 101, 111, 1011, 1101, \ldots\}$$

This correspond to the set of prime numbers $\{2, 3, 5, 7, 11, 13, \ldots\}$.

Sometimes it's convenient to use the same name for the concrete problem and its associated language:

$$\textsc{Prime} = \{10, 11, 101, 111, 1011, 1101, \ldots\}$$

Definitions

An **alphabet** Σ is any finite set of symbols.

A **language** L over Σ is any set of strings made up of symbols from Σ.

We denote the empty string by ϵ, and the empty language by $\emptyset$. The language of all strings over Σ is denoted by Σ^*. So, if $\Sigma = \{0,1\}$, then

$$\Sigma^* = \{\epsilon, 0, 1, 00, 01, 10, 11, 000, \ldots\}$$

is the set of all binary strings.

Every language L over Σ is a subset of Σ^*.

Operations on Languages:

operation:	meaning:
union of L_1 and L_1	$L_1 \cup L_2$
intersection of L_1 and L_2	$L_1 \cap L_2$
complement of L	$\overline{L} = \Sigma^* - L$
concatenation of L_1 and L_2	$L_1 L_2 = \{x_1 x_2 : x_1 \in L_1 \text{ and } x_2 \in L_2\}$
Kleene star of L	$L^* = \{\epsilon\} \cup L \cup L^2 \cup L^3 \cup \ldots$ where $L^k = LL \ldots L$, k times.

Algorithms and Turing Machines

We have already formalized the notion of **problem** by associating a **formal language** with it.

Now, we need to formalize the notion of **algorithm**.

In order to solve a problem, represented by its associated language, an algorithm amounts to simply checking if the input string belongs to the language.
You may have studied some **computational model** that operates on strings and **accepts** a language.

Examples of such models are **finite state automata** and **pushdown automata**.
These models are good for recognizing certain languages, but there are many languages that are not recognizable by them. For example:

The language $L = \{0^n 1^n : n \geq 0\}$ cannot be accepted (recognized) by any finite state automaton.
The language $L = \{0^n 1^n 2^n : n \geq 0\}$ cannot be accepted (recognized) by any pushdown automaton.

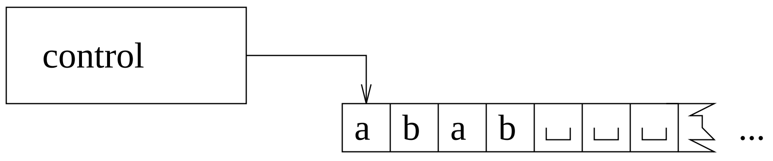
A **Turing Machine** is a very simple model, similar to a finite state automaton, that can simulate "arbitrary algorithms".

The description of a Turing machine

The Turing machine contains:

- an infinite tape (representing unlimited memory);
- a tape head capable of reading and writing symbols and of moving around the tape;
- a control implementing a transition function: given the current machine configuration (its machine state, tape contents and head position) the control changes the machine to another configuration using a "single step".

Initially, the tape contains only the input and is blank everywhere else; the head of the machine is on the leftmost position. Information can be stored by writing it on the tape. The machine keeps "computing" (moving from configuration to configuration) until it decides to produce an output. The outputs are either **accept** or **reject**. It could possibly go on forever.



Example

Describe a Turing Machine that recognizes the language
$A = \{0^{2^n} : n \geq 0\} = \{0, 00, 0000, 00000000, \ldots\}$.

```
On input string w do:
1. Sweep left to right across the tape,
   crossing off every other 0.
2. If in stage 1 the tape contained a single 0, accept.
3. If in stage 1 the taped contained more than a
   single 0 and the number of zeros was odd, reject.
4. Return the head to the left-hand end of the tape.
5. Go to stage 1.
```

A sample run of the above machine on input `0000`.
The tape is initially: (the thick line shows the head position)

‖ 0	0	0	0	⊔	⊔	⊔···

The first zero is erased in order to "mark" the beginning of the tape:

⊔	‖ 0	0	0	⊔	⊔	⊔···

At the end of the first run of stage 1, the tape is:

⊔	x	0	x ‖	⊔	⊔ ⊔···

At the end of the second run of stage 1, the tape is:

⊔	x	x	x ‖		⊔	⊔···

At the end of the third run of stage 1, the tape is:

⊔	x	x	x ‖	⊔	⊔	⊔···

The machine **accepts** since the first blank represents a 0 in the tape.

Formal definition of Turing machine

A **Turing machine** is a 7-tuple, $(Q, \Sigma, \Gamma, \delta, q_1, q_{accept}, q_{reject})$, where Q, Σ, Γ are all finite sets and

1. Q is a set of states,
2. Σ is the input alphabet not containing the special **blank** symbol $\sqcup$,
3. Γ is the tape alphabet, where $\sqcup \in \Gamma$ and $\Sigma \subseteq \Gamma$,
4. $\delta : Q \times \Gamma \longrightarrow Q \times \Gamma \times \{L, R\}$ is the transition function,
5. $q_1 \in Q$ is the start state,
6. $q_{accept} \in Q$ is the accept state,
7. $q_{reject} \in Q$ is the reject state, where $q_{reject} \neq q_{accept}$.

A **configuration** of the Turing machine consists of the current state, the current head position and the current tape contents.

We can compactly represent a configuration, by writing the state exactly where the tape is.
For the previous machine, $\Sigma = \{0, 1\}$, $\Gamma = \{0, 1, \sqcup, \mathrm{x}\}$, and the first few configurations are:
$q_1 0000 \quad \sqcup q_2 000 \quad \sqcup \mathrm{x} q_3 00 \quad \sqcup \mathrm{x} 0 q_4 0 \quad \cdots$

Also,
$\delta(q_1, 0) = (q_2, \sqcup, R)$, $\delta(q_2, 0) = (q_3, \mathrm{x}, R)$, $\delta(q_3, 0) = (q_4, 0, R)$.

We say that $\cdots aq_ib\cdots$ **yields** $\cdots acq_j\cdots$, if $\delta(q_i, b) = (q_j, c, R)$.
Similarly, $\cdots aq_ib\cdots$ **yields** $\cdots q_jac\cdots$, if $\delta(q_i, b) = (q_j, c, L)$.

A Turing machine M **accepts** input w if a sequence of configurations $C_1, C_2, \ldots, C_k$ exists where:

1. C_1 is the start configuration q_1w,
2. each C_i yields C_{i+1}, and
3. C_k is an accepting configuration (a configuration with state being q_{accept}).

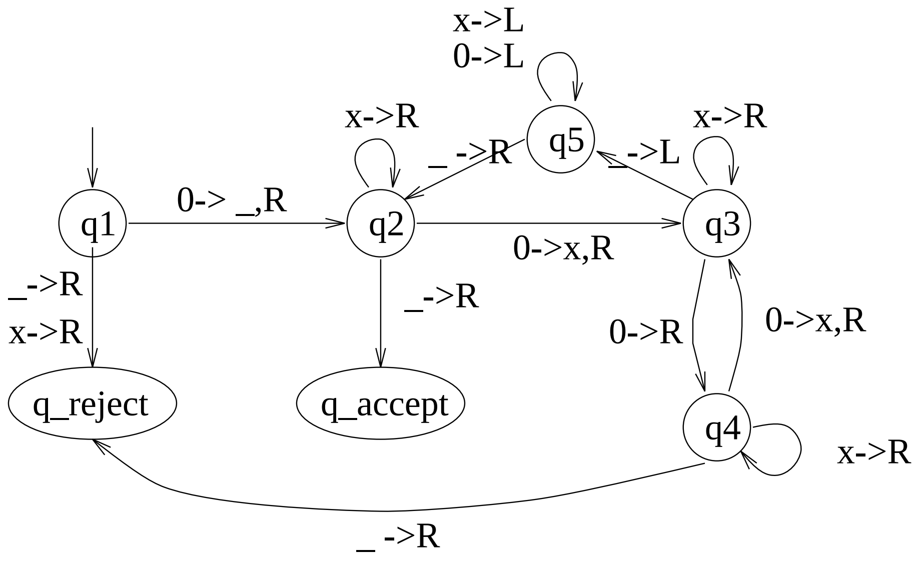
The collection of strings that M accepts is **the language of** M, denoted by $L(M)$.

A rejecting configuration is a configuration whose state is q_{reject}.

Three outcomes are possible when we start a Turing machine on an input: the machine may **accept**, **reject** or **loop** (not halt). So, for $w \notin L(M)$, M can fail to accept w by either entering the state q_{reject}, or by looping.

Turing machines that halt in all inputs are called **deciders**. A **decider** that recognizes some language is said to **decide** that language.

A decider for $A = \{0^{2^n} : n \geq 0\}$



_ is $\sqcup$

A simple run on input 0000:

(q1) 0000
_(q2)000
_x(q3)00
_x0(q4)0
_x0x(q3)_
_x0(q5)x_
_x(q5)0x_
_(q5)x0x_
(q5)_x0x_
_(q2)x0x_
_x(q2)0x_
_xx(q3)x_
xxx(q3)
_xx(q5)x_
_x(q5)xx_
(q5)xxx
(q5)_xxx_
(q2)xxx
_x(q2)xx_
_xx(q2)x_
xxx(q2)
xxx(q-accept)

Algorithms are Turing machines

Algorithms have been used for centuries, for example: the division algorithm learned in school, Euclid's algorithm for calculating the greatest common divisor of two integer numbers.

Informally, an **algorithm** is a collection of simple instructions for carrying out some task.
In 1900, Hilbert posed a problem (Hilbert's tenth problem) that asked for an **algorithm**, although this word was not used but: "a process according to which it [the solution to the problem] can be determined by a finite number of operations."

The formal definition came in 1936. Alan Turing used Turing machines to define algorithms. Independently, Alonzo Church used a notational system called λ-calculus in his definition of algorithms. These two definitions were shown to be equivalent. The connection between the informal notion of algorithm and the precise definition is called the **Church-Turing thesis.**

Turing showed that Hilbert's tenth problem is **undecidable**, that is, that there exists no Turing machine that decides it, no algorithm that halts in all inputs that solves the problem. The **halting problem** was also shown to be undecidable.

There are several **models of computation** that can be used to describe algorithms:

- Turing machines,
- multitape Turing machines,
- random access machine (RAM) programs,
- non-deterministic Turing machines, etc.

(Programs using the RAM model (random access machine) are quite similar to programs in real computers)

They all share the following characteristics:

- If a language is decidable in one model, it is also decidable in the other models.
  This means they are equivalent with respect to **computability theory**.
- If a language can be decided in a polynomial number of basic steps in one of the first 3 models, it can also be decided in a polynomial number of basic steps in the other models.
  This means that the first 3 models are equivalent in terms of **complexity theory**.

So, for studying NP-completeness, it is reasonable to define:

Problems are **languages** over a **finite alphabet**.
Algorithms are **Turing machines**.

Coming up next: time complexity classes for Turing machines; other models of computation.

ANALYSIS OF ALGORITHMS AND COMPLEXITY CLASSES

Analysis of Algorithms

The number of steps that an algorithm takes on an input depends on the particular input.

Example: How many comparisons are done in `bubblesort` in the following cases:

- Sorted array with 10 elements:

- Array with 10 elements in reverse order:

For simplicity, we prefer to measure the running time as a function of the size of the input. For this reason, we use **worst-case analysis** in which the running time for inputs of a given size is taken to be the longest running time among all inputs of that size.

- Using **worst-case analysis** for `bubblesort`, we would say that the running time (measured here in number of comparisons) on arrays with 10 elements is:

In **average-case analysis** the running time is measured as the average of all the running times of inputs of a particular length.

Example: The worst-case running time for `quicksort` is proportional to n^2, while its average-case running-time is proportional to $n \log n$, for arrays of size n.

Analyzing Turing machine algorithms

We have identified **problems** with **formal languages** and **algorithms** with **Turing machines**.

We must define **running time** for this model of computation:

Definition 1 *Let M be a deterministic Turing machine that halts on all inputs. The* **worst-case running time** *or* **time complexity** *of M is the function $T : \mathcal{N} \rightarrow \mathcal{N}$, where $T(n)$ is the maximum number of steps that M uses on any input of length n. If $T(n)$ is the worst-case running time of M, we say that M runs in time $T(n)$ and that M is an $T(n)$ time Turing machine.*

The running time of an algorithm is often a complex expression, so we usually just estimate it using **asymptotic analysis**.
For example, if the worst-case running time of a Turing machine M is

$$T(n) = 7n^3 + 10n^2 + 3n + 100$$

we just look at the highest order term $7n^3$ and disregard the constant saying that

$$T(n) \text{ is of the order of } n^3$$

or that

n^3 is an **asymptotic upper bound** on T(n).

Review of Big-O, small-o, Ω and Θ notation

We formalize the previous notion using big-O notation.

Given a function $g(n)$ we denote by $O(g(n))$ the set of functions:

$$O(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \le f(n) \le cg(n) \text{ for all } n \ge n_0\}$$

When $f(n) \in O(g(n))$, often written as $f(n) = O(g(n))$, we say that $f(n)$ is in big-Oh of $g(n)$ and that $g(n)$ is an **asymptotic upper bound** for $f(n)$.
Using this notation, we can say that
$T(n) = 7n^3 + 10n^2 + 3n + 100 \in O(n^3)$.

Why?

Give c and n_0 such that $7n^3 + 10n^2 + 3n + 100 \le cn^3$ for all $n \ge n_0$. There are many possible values.

$$c = \qquad\qquad n_0 =$$

Proof:

The Ω notation provides an **asymptotic lower bound** on a function. Given a function $g(n)$ we denote by $\Omega(g(n))$ the set of functions:

$$\begin{aligned}\Omega(g(n)) = \{f(n) :\ & \text{there exist positive constants } c \text{ and } n_0 \\ & \text{such that } 0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0\}\end{aligned}$$

Using this notation, we can say that
$T(n) = 7n^3 + 10n^2 + 3n + 100 \in \Omega(n^3)$.

The Θ notation provides an **asymptotic tight bound** on a function. Given a function $g(n)$ we denote by $\Theta(g(n))$ the set of functions:

$$\Theta(g(n)) = O(g(n)) \cap \Omega(g(n))$$

Let $T(n) = 7n^3 + 10n^2 + 3n + 100$.
Since $T(n) \in O(n^3)$ and $T(n) \in \Omega(n^3)$ then $T(n) \in \Theta(n^3)$.
The asymptotic upper bound given by the O-notation may or may not be asymptotically tight:

- the bound $2n^2 \in O(n^2)$ is asymptotically tight,
- the bound $2n \in O(n^2)$ is not asymptotically tight.

The little-oh notation is used to define an upper bound that is not asymptotically tight. Given a function $g(n)$ we denote by $o(g(n))$ (little-oh of $g(n)$) the set of functions:

$$\begin{aligned}o(g(n)) = \{f(n) :\ & \text{for any constant } c > 0, \text{ there exists a constant} \\ & n_0 > 0 \text{ such that } 0 \leq f(n) < cg(n) \text{ for all } n \geq n_0\}\end{aligned}$$

Thus, $2n \in o(n^2)$ but $2n^2 \notin o(n^2)$

To review the previous concepts, refer to Chapter 2 of the textbook by Cormen, Leiserson and Rivest.

Solve the following exercise for next class.

Write **`yes`** or **`no`** on the cells of the table, depending on whether the function in a given row is in the set given in the column. Give a proof of your answer for the cells marked with *.

Is $f(n) \in$... ?	$o(n)$	$O(n)$	$\Theta(n)$	$O(n^2)$	$\Theta(n^2)$	$O(2^n)$	$\Omega(2^n)$
$10 \log n$	*						
$n \log n$							
$5n$							
$1000n^2$					*		
$2^{\log_2 n}$		*					
2^{n+1}							
n^n						*	*

Analysing a Turing machine algorithm

Let us analyse an algorithm for deciding $A = \{0^k 1^k : k \geq 0\}$.

```
M1="On input string w do:
  1. Scan across the tape and reject if a 0 is found
     to the right of a 1.
  2. Repeat the following if both 0s and 1s remain on
     the tape:
  3.    Scan across the tape, crossing off a single 0
        and a single 1.
  4. If 0s still remain after 1s have been crossed off,
     or if 1s still remain after all the 0s have been
     crossed off, reject. Otherwise, if neither 0s nor
     1s remain on the tape, accept."
```

The scan in **stage** `1.` plus the re-positioning of the head at the beginning takes $O(n)$ steps.
In **stages** `2.` and `3.` the machine repeatedly double-scans the tape, once to check the condition in `2.` and once to cross off a 0 and a 1. Since each scan crosses off 2 symbols, at most $n/2$ double-scans will occur. Since each double-scan takes $O(n)$ steps, the total time for stages `2.` and `3.` is $O(n^2)$ steps.
In **stage** `4.` a last scan is done on the tape, taking $O(n)$ steps.
So, the worst-case running time for `M`, denoted by $T(n)$, is in $O(n) + O(n^2) + O(n) = O(n^2)$.

Homework: draw a state diagram for `M` *for handing in next class, various different implementations are possible.*

Analysing another Turing machine algorithm

Recall the Turing machine `M2` that decides $A = \{0^{2^l} : l \geq 0\}$.

```
M2="On input string w do:
0. Sweep left to right, if any symbol is not 0, reject.
1. Sweep left to right across the tape,
   crossing off every other 0.
2. If in stage 1 the tape contained a single 0, accept.
3. If in stage 1 the tape contained more than a
   single 0 and the number of zeros was odd, reject.
4. Return the head to the left-hand end of the tape.
5. Go to stage 1."
```

Let $T(n)$ be the worst-case running time of `M1` on inputs of size n.
Recall that we are measuring basic steps of the machine.
One execution of `0.` or `1.` takes $O(n)$ steps.
The conditions verified in `2.` and `3.` are implemented as a single transition from q_2 to q_{accept} and from q_4 to q_{reject}, respectively.
So, `2.` and `3.` run in $O(1)$ steps.
The algorithm halves the number of zeros on the tape at each execution of `1.` If we pass step 0., the tape contains n 0's at the beginning. Let k be such that $2^{k-1} < n \leq 2^k$, or equivalently $k = \lceil log_2 n \rceil$. After j executions of `1.`, the number of zeroes is at most $\frac{2^k}{2^j}$, so if the algorithm has not halted before, in at most k executions of `1.` there will be a single 0 on the tape and it will halt in `2.` So, `1.-3.` are executed at most $O(\log n)$ times, and therefore $T(n) \in O(n \log n)$.

Time complexity classes

The following notation is used for classifying languages according to their time requirement:
Let $t : \mathcal{N} \to \mathcal{N}$ be a function. Define the **time complexity class**, TIME($t(n)$), to be

$$\text{TIME}(t(n)) = \{L : L \text{ is a language decided by an } O(t(n)) \text{ time Turing machine}\}$$

Turing machine `M1`, which decides $A = \{0^k 1^k : k \geq 0\}$, runs in time $O(n^2)$.
Therefore, $A \in \text{TIME}(n^2)$.

Let us consider a language L for which we can construct:

- an $O(n^3)$ Turing machine M_1
- an $O(n \log n)$ Turing machine M_2

Is $L \in \text{TIME}(n \log n)$? why?

Is $L \in \text{TIME}(n^3)$? why?

Is $L \in \text{TIME}(n^2)$? why?

Multitape Turing Machines

A **multitape Turing machine** is like an ordinary Turing machine but it has several tapes.
Each tape has its own reading/writing head.
Initially the input appears on tape 1 and all other tapes start blank.

The transition function is changed in order to allow manipulation of all tapes simultaneously. Note that the transition still goes from one state to another.
Let k be the number of tapes on the machine, then the transition function is defined as

$$\delta : Q \times \Gamma^k \to Q \times \Gamma^k \times \{L, R\}^k$$

Let us consider a 3-tape Turing Machine ($k = 3$). Then,

$$\delta(q_2, a, b, d) = (q_3, b, a, a, R, R, L)$$

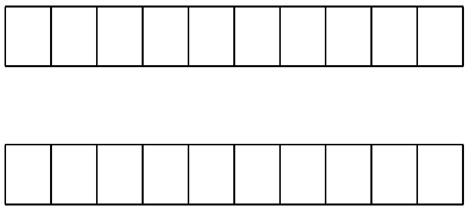
means that if the machine is in state q_2, head 1 is reading symbol a, head 2 is reading symbol b and head 3 is reading symbol d, then the machine goes to state q_3, writes symbol b on tape 1, writes symbol a on tapes 2 and 3 and moves head 1 and 2 to the right and head 3 to the left.

A two-tape Turing Machine

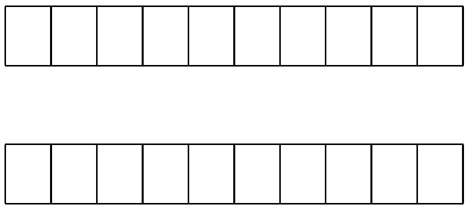
Let $A = \{0^k 1^k : k \geq 0\}$. We have shown that this language can be decided by a $O(n^2)$ single-tape Turing machine.

Give an $O(n)$ two-tape Turing machine for deciding A.

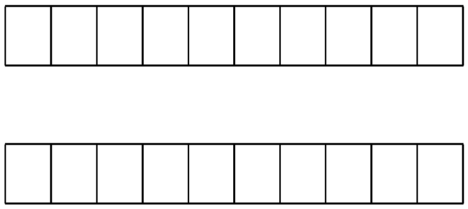
Simulating this two-tape Turing machine for string `000111`:



Simulating this two-tape Turing machine for string `0001111`:



Simulating this two-tape Turing machine for string `0000111`:



A linear time two-tape machine for $A = \{0^k 1^k : k \geq 0\}$

```
M3= "On input string w:
    1. Scan across tape 1 and if a 0 is found to the
       right of a 1, then reject.
    2. Scan across the 0s on tape 1, simultaneously
       copying each onto tape 2, until the first 1
       appears on tape 1.
    3. Scan across the 1s on tape 1, simultaneously
       crossing off each 0 on tape 2 for each 1 on
       tape 1. If all 0s in tape 2 are crossed off
       before all the 1s are read from tape 1, reject.
    4. If all the 0s have now been crossed off, accept.
       Otherwise, reject."
```

Analysing the the time complexity of `M3`:

Stage `1.` runs in $O(n)$ steps.
Stages `2.`, `3.` and `4.` combined take $O(n)$ steps.
Therefore, `M3` is an $O(n)$ two-tape Turing machine.

Obs: if the second statement about stages `2.`, `3.` *and* `4.` *is not clear, try to write the state diagram for this machine, to be convinced that the 3 steps can be accomplished with a single pass.*

Complexity relationships among models

Now we examine how the number of tapes in a Turing machine can affect the time complexity of languages.

Theorem 1 *Let $t(n)$ be a function, where $t(n) \geq n$. Then, every $t(n)$ time multitape Turing machine has an equivalent $O(t^2(n))$ time single-tape Turing machine.*

Proof idea:
The main idea is to simulate a k-tape Turing machine on a single-tape one, using the following tricks:

- Keep track of the contents of the k tapes in a single tape with a special character, say `#`, as a delimiter.
- Keep track of the position of the k heads, by using a new special symbol for each symbol on the tape alphabet of the multitape machine. For instance `a` and $\bar{\mathtt{a}}$ both correspond to symbol `a`, but $\bar{\mathtt{a}}$ indicates the head of a tape is over this symbol.
- If at any point the original machine moves its head on the unread portion of some tape, the single-tape machine will write a blank on top of the corresponding `#` and shift all the tape contents until the last `#` one position to the right. Then, continue simulating as before.

Proof:

Let M be a k-tape Turing machine that runs in time $t(n)$. We construct a single tape Turing machine S to simulate M in the way described before.
For each step of machine M, machine S does various steps:

- One scan to know the contents of the symbols under all the tape heads in order to determine the next move.
- One second pass to update tape contents and head positions.
- At most k right shifts of the tape contents.

Each of these operations will take time proportional to the active part of S's tape, so we need to estimate an upper bound on this.
For each step of M at most one extra position becames active on each tape (write on the blank portion of the tape). Since there are k tapes, and originally $n+k$ positions were active, after $t(n)$ steps at most $n+k+kt(n)$ positions are active, which is in $O(t(n))$.
All the stages of the simulation of one of M's steps takes time proportional to the size of the active part of S's tape, yielding $2O(t(n)) + kO(t(n)) = O(t(n))$ steps in order to simulate each single step of M. Since the total number of steps is $O(t(n))$, then the total time for S is $O(t(n))O(t(n)) = O(t^2(n))$.

Random Access Machines (RAM model)

A **random access machine**, or RAM, is a computing device containing an infinite array of registers, each able to hold an *arbitrarily large integer*. Register i, $i \geq 0$, is denoted r_i.
The input of a RAM program is a finite sequence of integers $I = (i_1, i_2, \ldots, i_n)$.
A RAM program $\Pi = (\pi_1, \pi_2, \ldots, \pi_m)$ is a finite sequence of instructions, where each instruction π_i is one of:

Instruction	**Operand**	**Semantics**
READ	j	$r_0 := i_j$
READ	$\uparrow j$	$r_0 := i_{r_j}$
STORE	j	$r_j := r_0$
STORE	$\uparrow j$	$r_{r_j} := r_0$
LOAD	x	$r_0 := x$
ADD	x	$r_0 := r_0 + x$
SUB	x	$r_0 := r_0 - x$
HALF		$r_0 := \lfloor \frac{r_0}{2} \rfloor$
JUMP	j	$\kappa := j$
JPOS	j	**if** $r_0 > 0$ **then** $\kappa := j$
JZERO	j	**if** $r_0 = 0$ **then** $\kappa := j$
JNEG	j	**if** $r_0 < 0$ **then** $\kappa := j$
HALT		$\kappa := 0$

x can be: "j" (r_j), "$\uparrow j$" (r_{r_j}) or "$= j$" (integer j);
r_0 is the accumulator where all operations take place;
κ is the "program counter".

Unless specified otherwise, at the end of the execution of an instruction the program counter is automatically updated: $\kappa := \kappa + 1$.
When the program halts, the output is the content of r_0.
If we focus on decision problems, then the output is either *1* (accept) or *0* (reject).

A *configuration* of a RAM is a pair $C = (\kappa, R)$, where κ is the instruction to be executed, and $R = \{(j_1, r_{j_1}), (j_2, r_{j_2}), \ldots, (j_k, r_{j_k})\}$ is a finite set of register-value pairs (all other registers are zero).
Let D be a set of finite sequences of integers and let $\phi : D \rightarrow \{0, 1\}$ be the function to be computed by a RAM program Π.
The initial configuration is $(1, \emptyset)$ and for an input $I \in D$, the final configuration is $(0, R)$ where $(0, \phi(I)) \in R$.

Input size is measured as the number of bits required to store the input sequence. For an integer i let $b(i)$ be its *binary representation* with no redundant leading 0s, and with a minus in front, if negative. If $I = (i_1, \ldots, i_k)$ is a sequence of integers, its length is defined as $l(I) = \Sigma_{j=1}^{k} |b(i_j)|$.

Each RAM operation is counted as a **single step**, even though it operates on arbitrarily large integers.
Let $f : \mathcal{N} \rightarrow \mathcal{N}$ be a function, and suppose that for any $I \in D$ the RAM program computes ϕ in k steps where $k \leq f(l(I))$.
Then, we say that Π **computes** ϕ in time $f(n)$.

The RAM model seems much more powerful than Turing machine models, since in a single step we can operate on arbitrarily large integers.

We will show the expected fact that any Turing machine can be simulated by a RAM program with no loss of efficiency.

We will also show the rather **surprising fact** that any RAM program can be simulated by a Turing machine with only a **polynomial** loss of efficiency!!!

We have to adjust the parameters of the Turing machine to correspond to the ones of a RAM program.
Suppose that $\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_k\}$ is the alphabet of a Turing machine. Then, let
$D_\Sigma = \{(i_1, i_2, \ldots, i_n, 0) : n \geq 0, 1 \leq i_j \leq k, j = 1, \ldots, n\}$.
If L is a language over Σ, define $\phi_L : D_\Sigma \to \{0, 1\}$ with

$$\phi_L(i_1, i_2, \ldots, i_n, 0) = \begin{array}{ll} 1, & \text{if } \sigma_{i_1}\sigma_{i_2}\ldots\sigma_{i_n} \in L, \\ 0, & \text{otherwise.} \end{array}$$

(Computing ϕ_L in the RAM model is the same as deciding L in the Turing machine model).

Note that the last 0 in the input helps the RAM program "sense" the end of the Turing machine's input.

RAM program simulating a Turing machine

Theorem 2 *Let L be a language in TIME$(f(n))$. Then there exists a RAM program which computes function ϕ_L in time $O(f(n))$.*

Proof:

Let $M = (Q, \Sigma, \Gamma, \delta, q_1, q_{accept}, q_{reject})$ be the Turing machine that decides L in time $f(n)$.

Our RAM program copies the input string to registers $2, 3, \ldots, n+1$. Register 1 will point to the currently scanned symbol, so initially $r_1 := 2$.

From now on, the program simulates the steps of the Turing machine one by one.

Let $\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_k\}$. A set of instructions simulates each state $q \in Q$; their instruction numbers are:

$$\begin{array}{llll}
N_{q,\sigma_1}, & N_{q,\sigma_1}+1, & \ldots, & N_{q,\sigma_1}+9 \\
N_{q,\sigma_2}, & N_{q,\sigma_2}+1, & \ldots, & N_{q,\sigma_2}+9 \\
\vdots & \vdots & \vdots & \vdots \\
N_{q,\sigma_k}, & N_{q,\sigma_k}+1, & \ldots, & N_{q,\sigma_k}+9
\end{array}$$

So, if $\delta(q, \sigma_j) = (p, \sigma_l, dir)$, then the instructions corresponding to this transition are:

N_{q,σ_j}.	LOAD $\uparrow 1$	(fetch symbol under the head)
$N_{q,\sigma_j}+1$.	SUB $= j$	(j is the symbol index)
$N_{q,\sigma_j}+2$.	JZERO $N_{q,\sigma_j}+4$	(if symbol=σ_j we have what to do)
$N_{q,\sigma_j}+3$.	JUMP $N_{q,\sigma_{j+1}}$	(otherwise try next symbol)
$N_{q,\sigma_j}+4$.	LOAD $= l$	(σ_l is the symbol to be write)
$N_{q,\sigma_j}+5$.	STORE $\uparrow 1$	(write σ_l)
$N_{q,\sigma_j}+6$.	LOAD 1	(the head position)
$N_{q,\sigma_j}+7$.	ADD $= d$	(d is 1 if $dir = R$ and -1 if $dir = L$)
$N_{q,\sigma_j}+8$.	STORE 1	(update head position)
$N_{q,\sigma_j}+9$.	JUMP N_{p,σ_1}	(start simulation of state p)

The states q_{accept} and q_{reject} are simulated as:

$N_{q_{accept}}$.	LOAD $= 1$	($r_0 := 1$)
$N_{q_{accept}}+1$.	HALT	
$N_{q_{reject}}$.	LOAD $= 0$	($r_0 := 0$)
$N_{q_{reject}}+1$.	HALT	

The time needed to execute the instructions simulating a single step of the Turing machine is a constant. Thus, the RAM pogram needs to execute only $O(f(n))$ instructions. $\square$

Turing machine simulating a RAM program

Let $\phi : D \to \{0, 1\}$ be a function. Then, define L_ϕ to be a language over a finite alphabet, say $\Sigma = \{\mathtt{0}, \mathtt{1}, \#, -\}$, such that $b(i_1)\#b(i_2)\#\cdots\#b(i_l) \in L$ if and only if $\phi((i_1, i_2, \cdots, i_l)) = 1$.

Theorem 3 *Let* Π *be a RAM program that computes* ϕ *in time* $f(n) \geq n$*. Then, there exists a 7-tape Turing machine which decides* L_ϕ *in time* $O(f^3(n))$*.*

Proof:
The Turing machine M will simulate Π using 7 tapes. The tape contents are as follows:

tape 1:	the input string
tape 2:	the representation of R the register contents (it is a sequence of strings of the form $b(i) : b(r_i)$, separated by ; and possibly blanks, with an endmarker)
tape 3:	value of κ, the program counter
tape 4:	current register address
tape 5:	holds one of the operands for an arithmetic operation
tape 6:	holds the other operand for an arithmetic operation
tape 7:	holds the result of an arithmetic operation

Note that every time a register is updated, we may have to shift the contents of other registers to the right to accommodate the new value. This can be done in time proportional to the length of tape 2.

The states of M are subdivided into m groups, where m is the number of instructions in Π. Each group implements one instruction of Π.

We will show the following claim:
"After the tth step of a RAM program computation on input I, the contents of any register have length at most $t + l(I) + l(B)$, where B is the largest integer referred to in an instruction of Π."

Assume this claim is correct.
We only need to show that simulating an instruction of Π by M takes $O(f^2(n))$. Decoding Π's current instruction takes constant time. Fetching the value of registers, involved in an instruction, takes $O(f^2(n))$ time, since tape 2 contains $O(f(n))$ pairs, each has length $O(f(n))$ (by the claim) and searching can be done in linear time on the size of the tape. The computation of the result itself involves simple arithmetic functions (ADD, SUB, HALF) on integers of length $O(f(n))$, which can be done in $O(f(n))$.

Since each of Π's instructions can be simulated in $O(f^2(n))$, all $f(n)$ instructions can be simulated in time $O(f^3(n))$ by the Turing machine.

$\square$

Note that we still have to prove the claim above...

CLAIM: "After the tth step of a RAM program computation on input I, the contents of any register have length at most $t + l(I) + l(B)$, where B is the largest integer referred to in an instruction of Π."

Proof:

The proof is by induction on t. The claim is true before the first step. Suppose the claim is true up to step $t - 1$.
We show it remains true after step t. We will check several cases of instructions.
If the instruction is a "jump" or HALT, then there is no change on the contents of the registers.
If it is a LOAD or STORE that modifies a register, its contents was at another register at a previous step, so the claim holds.
If it was a READ then $l(I)$ guarantees the claim.
Finally, if it is an arithmetic operation, say ADD, it involves the addition of two integers, i and j. Each is either contained in a register of last step, or it is a constant mentioned in Π, and so its size is not bigger than $l(B)$. The length of the result is at most one plus the length of the longest operand which, by induction, is at most $t - 1 + l(I) + l(B)$. The situation with SUB is identical and with HALF is easier.
$\square$

Note that $t + l(I) + l(B) \in O(f(n))$, since $t \leq f(n)$, $l(I) = n \in f(n)$ and $l(B) \in O(1)$.

$\boxed{\text{The complexity class P}}$

Summary of polynomial equivalences between models
Suppose the algorithm to be simulated in one of the models takes $f(n)$ steps on an input of size n.
The following table shows the running time taken by the simulator machine.

$\downarrow$ simulates $\rightarrow$	single-tape Turing machine	multitape Turing machine	RAM program
single tape Turing machine	-	$O(f^2(n))$	$O(f^6(n))$
multitape Turing machine	$O(f(n))$	-	$O(f^3(n))$
RAM program	$O(f(n))$	$O(f(n))$	-

We can now define the **complexity class P** as the set of concrete problems that are solvable in polynomial time by any of the above models.
Using the single-tape Turing machine model:
$\mathbf{P} = \cup_{k=0}^{\infty} \text{TIME}(n^k)$

Using the RAM model:
$\mathbf{P} = \{L : L \text{ can be decided by a polynomial time RAM algorithm}\}$

Note that the first one was the original definition, and the second one is the one used in most of the textbook by Cormen et al.

$\boxed{\text{Definition of the complexity class P}}$

For the purpose of the theory of NP-completeness, we can take our **algorithms** to be defined in any of the following computation models: single tape Turing machines, multitape Turing machines or random access machines. **The set of languages that can be decided in polynomial time is the same, regardless of which of the above models we use for defining algorithms.** The analysis of the algorithms in the textbook are based on the RAM model.
We will concentrate on the **binary alphabet**, although other alphabets could be used (remember that the length of strings representing integers in binary, ternary or decimal alphabets are all polynomial-time related).

Recall that an algorithm A **accepts** a string $x \in \{0,1\}^*$ if given input x, the algorithm outputs $A(x) = 1$. The language **accepted** by an algorithm A is $L = \{x \in \{0,1\}^* : A(x) = 1\}$. An algorithm A **rejects** input x if $A(x) = 0$. Even if a language L is accepted by an algorithm A, the algorithm will not necessarily reject a string $x \notin L$ (A may loop forever). A language L is said to be **decided** by an algorithm A if $A(x) = 1$ for all $x \in L$ and $A(x) = 0$ for all $x \in \{0,1\}^* - L$.
Thus, we can precisely define the complexity class P as:

$$\mathbf{P} = \{L \subseteq \{0,1\}^* \;:\; \text{there exists an algorithm } A \text{ that } \textbf{decides } L \text{ in polynomial time}\}$$

Accepting versus Deciding in polytime

The complexity class P was defined as the set of languages that can be **decided** in polynomial time.
We ask ourselves: is there a language that can be accepted by a polynomial time algorithm, but cannot be decided by a polynomial time algorithm?
The following theorem shows the answer to this question is "no":

Theorem 4

Let $P' = \{L \subseteq \{0,1\}^* \ : \text{ there exists an algorithm } A \text{ that } \textbf{accepts } L \text{ in polynomial time}\}.$

Then, $\mathbf{P} = P'$.

Proof:
There are 2 sides to prove: $\mathbf{P} \subseteq P'$ and $P' \subseteq \mathbf{P}$.
By the definition of a language being accepted and decided by an algorithm, we can easily see that $\mathbf{P} \subseteq P'$. It remains to prove that $P' \subseteq \mathbf{P}$, that is, we must show that for any language L accepted by a polynomial-time algorithm, there exists a polynomial time algorithm that **decides** L.

Let L be a language such that there exists a polynomial-time algorithm A' that **accepts** L. Thus, A' accepts L in time $O(n^k)$ for some constant k, which implies that there exists a constant c such that A' accepts L in time at most $T(n) = cn^k$.
(Exercise: give a detailed proof of the last statement)

(proof, continued)

We define an algorithm A that decides L in the following way:

1. On an input x of size n, A simulates A' for the first $T(n) = cn^k$ or until A' halts on x, whichever comes first.
2. if A' reached an accepting configuration, then A outputs 1. Otherwise, if A' rejected L or after cn^k steps it did not halt, then ouput a 0.

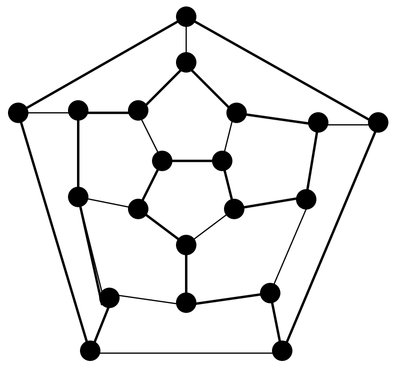
First, note that A really decides L. Indeed, if $x \in L$ then A' will accept it in at most cn^k steps, so an accepting configuration will be reached withing the number of steps A' was allowed to run, and A will output a 1. Moreover, if $x \notin L$, then the simulation of A' can either halt and reject or not halt up to step cn^k, in which cases, A will output a zero.
The second thing to verify is that A runs in polynomial time. This is clear, since in the first stage, A executes cn^k steps, for fixed constants c and k, and the second stage can be executed in a constant number of steps.
$\square$

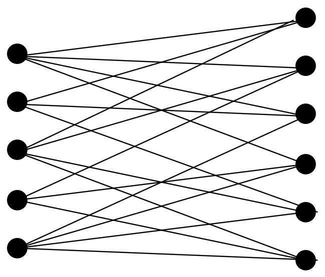
Exercise: Describe how A would simulate A' for T steps, both in the Turing machine model and in the RAM model.

The Hamiltonian-Cycle Problem

Let $G = (V, E)$ be an undirected graph with vertex-set V and edge-set E.
A **hamiltonian cycle** of G is a simple cycle (i.e. one with no repeated vertices) that contains each vertex in V.
A graph that contains a hamiltonian cycle is said to be a **hamiltonian** graph.

Example of a hamiltonian and a nonhamiltonian graph:





The problem of finding a hamiltonian cycle in an undirected graph has been studied for more than 100 years. Up to now, nobody was able to come up with a polynomial-time algorithm for this problem.
The formal language associated with the hamiltonian-cycle problem is defined as

$$\textsc{Ham-Cycle} = \{< G >: G \text{ is a hamiltonian graph } \}$$

Note that $< G >$ is a binary encoding of G, which uses either the adjacency matrix or the adjacency list representation.
Note: these representations are polynomial-time related. Why?

A polynomial-time verification algorithm

Even though we do not know any polynomial-time algorithm that accepts HAM-CYCLE, if somebody with psychic powers handed us a hamiltonian cycle, we could design a polynomial-time algorithm to verify whether the object handed to us was a valid hamiltonian cycle. This is the intuition behind **verification algorithms**.
We define a **verification algorithm** as a two-argument algorithm A, where one argument is an input string x and the other is a binary string y called a **certificate**. A two argument algorithm A **verifies** an input string x if there exists a certificate y such that $A(x, y) = 1$. The **language verified** by a verification algorithm A is

$$L = \{x \in \{0,1\}^* : \text{there exists } y \in \{0,1\}^* \text{ such that } A(x,y) = 1\}$$

In other words, an algorithm A verifies a language L if for any $x \in L$ there exists $y \in \{0,1\}^*$ such that $A(x,y) = 1$ and for any $x \notin L$ there exists no $y \in \{0,1\}^*$ such that $A(x,y) = 1$.

Let A be an algorithm that verifies a language L. We say that A verifies L **in polynomial time**, if there exists constants c_1, k_1, c_2, k_2 such that for any $x \in L$ of length n there exists a certificate y with $|y| \leq c_1 n^{k_1}$, such that $A(x,y) = 1$ and A runs in at most $c_2 n^{k_2}$ steps on arguments (x, y).
More intuitively: A is a polynomial-time algorithm and for each $x \in L$, there is a short certificate that allows A to verify (x, y) in polynomial time.

A poly-time verification algorithm for HAM-CYCLE

Let G be an undirected graph such that $< G > \in$ HAM-CYCLE. We wish to find a short certificate y for G and a verification algorithm that runs in polynomial-time on the length of $< G >$.
What is the approximate length of a graph $< G >$?
Let v be the number of vertices and e be the number of edges in G. If we use the adjacency-matrix representation, $| < G > | \in O(v^2)$, and if we use the adjacency-list representation,
$| < G > | \in O(v \cdot e \cdot \log v)$. Both representations are polynomially related since $v^2 \leq (v \cdot e \cdot \log v)^2$ and $v \cdot e \cdot \log v \leq v^4$, since $e \leq v^2$.
Give a short certificate for G and a polytime verification algorithm:
Let $h = (x_{i_1}, x_{i_2}, \ldots, x_{i_v})$ be a hamiltonian cycle in G (this exists since $< G > \in$HAM-CYCLE). Let $y =< h >$ be a binary encoding of h. Note that $|y| \in O(vlogv)$ which is polynomial on $| < G > |$.
The verification algorithm $A(< G >, < h >)$ is as follows:

```
$x_{i_{v+1}}$ := $x_{i_1}$;
for $k$ from 1 to $v$ do //check that h is a cycle
    if ($\{x_{i_k}, x_{i_{k+1}}\}$ is not an edge) then output 0;
    else mark $x_{i_k}$ as visited;
for $k$ from 1 to $v$ do //check that every vertex appeared
    if ($x_k$ was not visited) then output 0;
output 1;
```

For the adjacency-matrix representation, A runs in in $O(v)$ steps.
Exercise: analyse A for the adjacency-list representation.

Verification algorithms and the complexity class NP

Now, we are ready to define the complexity class **NP**. Intuitively, the complexity class **NP** is the class of languages that can be verified by a polynomial time algorithm.
More precisely,

$$\mathbf{NP} = \{\ L \in \{0,1\}^* : \text{there exists a two argument algorithm } A \text{ that } \textbf{verifies } L \textbf{ in polynomial time}\ \}.$$

Note: do not forget that the definition of an algorithm that "verifies a language in polynomial time" requires the existence of a short certificate.

By our previous discussion, HAM-CYCLE $\in$ **NP**.

Prove that $\mathbf{P} \subseteq \mathbf{NP}$:

Proof:
Let $L \in \mathbf{P}$. Thus, there exists a polynomial-time algorithm A' that decides L. Build a two-argument algorithm A such that for all $x, y \in \{0,1\}^*$, $A(x,y)$ simply returns $A'(x)$.
It is clear that for any $x \in L$ there exists a short certificate y (any short string will do) such that $A(x,y) = A'(x) = 1$. Moreover, since A' runs in polynomial time then so does A.
Therefore, A verifies L in polynomial time, and thus $L \in \mathbf{NP}$.

Nondeterministic Turing machines (NTMs)

The name **NP** comes from **nondeterministic polynomial time**, and is derived from an alternative characterization of **NP** which uses nondeterministic polynomial time Turing machines.

Let us first define nondeterministic Turing machines.

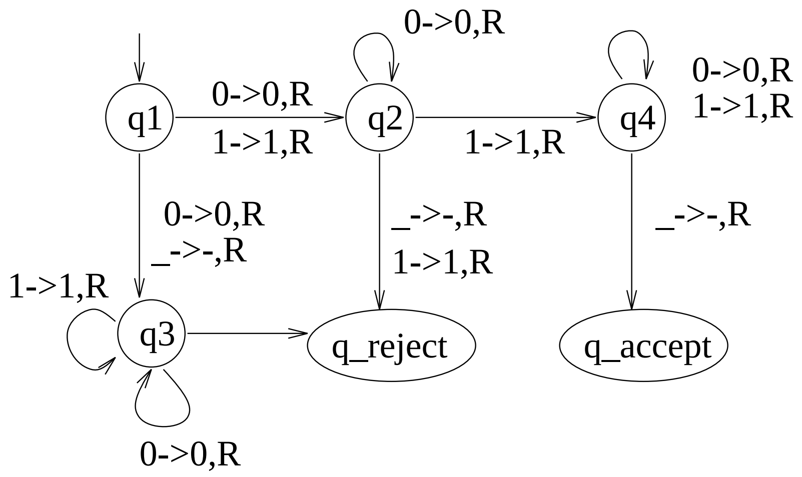
You may recall the difference between a nondeterministic finite state automaton (NFSA) and a deterministic one (DFSA). The NFSA may have several possible valid transitions when it reads a symbol, and a string is said to be accepted by the NFSA if there exists a sequence of valid transitions that ends up in a final state. A similar comparison can be done between nondeterministic Turing machines and deterministic ones.

A nondeterministic Turing machine is similar to a deterministic one, but at any point in a computation, the machine may proceed according to several possibilities. The transition function for a nondeterministic Turing machine has the form:

$$\delta : Q \times \Gamma \rightarrow \mathcal{P}(Q \times \Gamma \times \{L, R\})$$

The computation of a NTM is a tree whose branches correspond to different possibilities for the machine. We say that the machine **accepts** an in input if for **some** branch of the computation leads to the accept state.

Example of a nondeterministic Turing machine



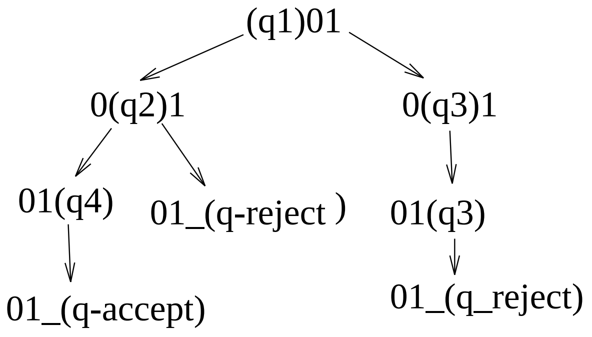
The nondeterminism can be seen in q_1 and q_2:

$$\begin{aligned}\delta(q_1,0) &= \{\{q_2,0,R\},\{q_3,0,R\}\}\\ \delta(q_2,1) &= \{\{q_{reject},1,R\},\{q_4,1,R\}\}\end{aligned}$$

`1` is rejected by N. The computation tree for `1` is:

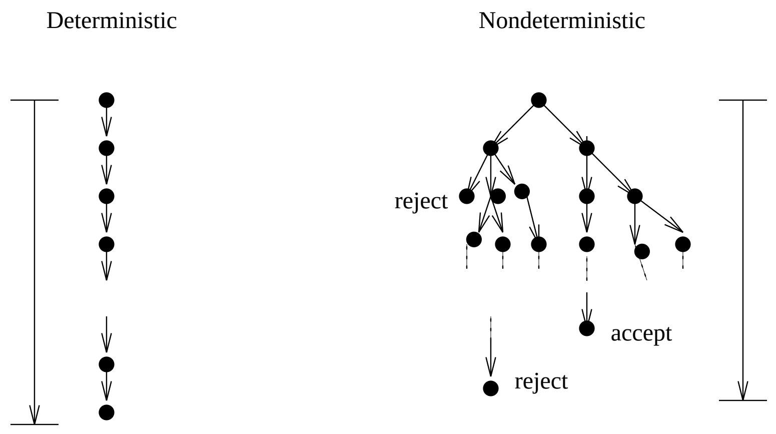
$q_1 1$
$1 q_2$
$1 \sqcup q_{reject}$

`01` is accepted by N. The computation tree for `01` is:



Measuring nondeterministic time

Definition 2 *Let N be a nondeterministic Turing machine that is a decider. The running time of N is a function $T(n)$, were $T(n)$ is the maximum number of steps that N uses on any branch of its computation on any input of length n.*



A NTM N runs in polynomial time if there exists a constant k such that its running time is in $O(n^k)$.

A polynomial-time NTM for HAM-CYCLE

```
N = `` On input $< G >$, where $G$ is an undirected
graph:

  1. Write a list of $v$ numbers $p_1, p_2, \ldots, p_v$, where $v$ is
     the number of vertices in $G$.  Each number in the
     list is nondeterministically selected to be
     between $1$ and $v$.

  2. Check for repetitions in the list.  If any are
     found, reject.

  3. For each $k$ between $1$ and $v-1$, check whether
     $\{p_k, p_{k+1}\}$ is an edge of $G$.  Check also if $\{p_v, p_0\}$
     is an edge.  If any are not, reject.  Otherwise,
     accept.''
```

This algorithm runs in nondeterministic polynomial time, since the nondeterministic selection in stage 1 runs in polynomial time and one can easily check that the other steps run in polynomial time.

You probably noticed the similarity between verification algorithms and nondeterministic Turing machines: the nondeterministic choice of the latter model is related to the certificate of the former one.

The polytime equivalence between verifiers and NTMs

Theorem 5 *A language is in* **NP** *if and only if it can be decided by some nondeterministic polynomial-time Turing machine.*

Proof:
$\rightarrow$ side:
Let $L \in \mathbf{NP}$, we must show that L can be decided by some polynomial time NTM. Let V be the polynomial time verification algorithm for L, which exists by the definition of **NP**. Let c and k be appropriate constants such that V can verify an input of size n in time cn^k, and construct a NTM as follows:

```
N = ‘‘On input $w$ of length $n$ do
  1. Nondeterministically select a string $y$ of length
     $cn^k$.
  2. Simulate $V$ on input $< w, y >$ for at most $cn^k$
     steps.
  3. If $V$ accepted the input, then accept; otherwise
     reject.
```

Note that a certificate that allows the verification of w by V under the given time bounds cannot exceed the size cn^k, so the above nondeterministic selection finds at least one such certificate for any w.

$\leftarrow$ side:
Let L be a language decided by a polynomial-time NTM N. We must construct a polynomial-time verification algorithm for L:

```
V = ``On input $< w, y >$ where $w$ and $y$ are strings:
  1. Simulate $N$ on input $w$, treating each symbol of $y$
     as a description of the nondeterministic choice
     to make at each step.
  2. If this branch of $N$'s computation accepts, then
     output 1; otherwise, output 0.''
```

Let $w \in L$. Note that since there exists a path on the computation tree of N that accepts w, then there exists y such that $V(w, y) = 1$. Moreover, since N runs in polynomial-time, then $|y|$ has polynomial size. Thus, V is a polynomial-time verification algorithm.