Let M be a deterministic Turing machine that halts on all inputs. The **running time** or **time complexity** of M is the function $f : \mathbb{N} \to \mathbb{N}$, where $f(n)$ is the maximum number of steps that M uses on any input of length n. If $f(n)$ is the running time of M, we say that M runs in time $f(n)$ and that M is an $f(n)$ time Turing machine. Customarily, we use n to represent the length of the input.

Let f and g be functions, $f, g : \mathbb{N} \to \mathbb{R}^+$. We say $f(n)$ is $\mathcal{O}(g(n))$ if positive integers c, n_0 exist such that for every integer $n \geq n_0$, $f(n) \leq cg(n)$.

Let f and g be functions, $f, g : \mathbb{N} \to \mathbb{R}^+$. We say $f(n)$ is $o(g(n))$ if for any real number $c > 0$, a number n_0 exists where $f(n) < cg(n)$ for all $n \geq n_0$

Analyzing Algorithms

Here is a Turing Machine to recognize $L = \{a^k b^k : k \geq 0\}$. The machine, M_1, on input string w:

1. Scan across and reject if any a right of a b
2. Repeat while any a, b still on tape:
3. Scan across, cross off one a, one b
4. If only one letter type remains, reject
   Otherwise, accept if neither a nor b remain

Question 1. What is the running time of M_1?

Let $t : \mathbb{N} \to \mathbb{R}^+$ be a function. Define the **time complexity class** TIME($t(n)$) be the collection of languages that are decidable by an $\mathcal{O}(t(n))$ time Turing machine.

Here is another machine, M_2, that decides L:

1. Scan across and reject if any a right of a b
2. Repeat as long as some of each remain:
3. If total $a + b$ is odd, reject.
4. Cross off every other a, every other b
5. If none of each remain, accept.

And here is yet another, M_3. Unlike M_1 and M_2, however, this one uses two tapes.

1. Scan across and reject if any a right of a b
2. Scan across the as on tape 1 until first b.
   While doing so, copy the as onto tape 2.
3. Cross off a and b, $1 : 1$ via two tapes
   If all a crossed off after and b remain, reject
4. If all a crossed off, accept. Else reject.

In computability theory (unit 3), all reasonable models are equivalent. In complexity theory, this is not the case: the computational model affects what running times we can achieve. Yet, the choice of deterministic models does not affect us too much. For example, we could prove the following relationship between multi-tape Turing Machines and single-tape Turing Machines:

Example: Let $t(n)$ be a function with $t(n) \geq n$. Show that every $t(n)$ time multi-tape Turing Machine has an equivalent $\mathcal{O}(t^2(n))$ time single-tape Turing Machine

Non-deterministic Running Time

Let N be a non-deterministic Turing Machine that is a decider. The **running time** of N is the function $f : \mathbb{N} \to \mathbb{N}$, where $f(n)$ is the **maximum number** of steps that N uses on any branch of its computation on any input of length n.

You might think of this as forking and we need every branch (thread, process, whatever) to terminate before we're done. Despite that analogy, this is not intended as a model of real world computation. You might want to think about why do we use the maximum number of steps on any branch instead of fewest, or the fewest that causes an accept?

The relationships between deterministic and non-deterministic running time, however, can be summarized as follows.

Let $t(n)$ be a function with $t(n) \geq n$. Show that every $t(n)$ time nondeterministic single-tape Turing Machine has an equivalent $2^{\mathcal{O}(t(n))}$ time deterministic single-tape Turing Machine

The proof of this involves analyzing the running time for a deterministic Turing Machine that simulates the branches of the non-deterministic one's computation tree.

The Class $\mathcal{P}$

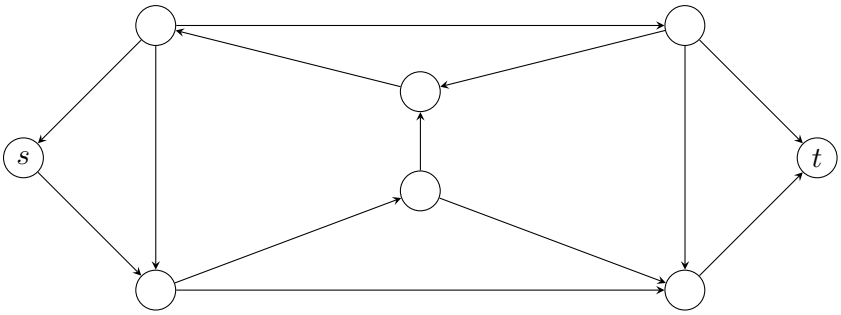
$\mathcal{P}$ is the class of languages that are decidable in polynomial time on a deterministic single-tape Turing machine. In other words,

$$\mathcal{P} = \bigcup_k \text{TIME}(n^k)$$

Question 2. Does this mean that polynomial differences are not important?

1. $\mathcal{P}$ is invariant for all models of computation that are polynomially equivalent to the deterministic single-tape Turing machine.

2. $\mathcal{P}$ roughly corresponds to the class of problems that are realistically solvable on a computer.



HAMPATH $= \{\langle G, s, t\rangle : G$ is a directed graph with a Hamiltonian path from s to $t\}$
Question 3. What class of running time is the best currently known algorithm to decide HAMPATH?

Question 4. What type of problem related to HAMPATH can we solve in polynomial time?

COMPOSITES = $\{x : x = pq$, for integers $p, q > 1\}$
Question 5. What do you need for a verifier?

Question 6. Is this polynomially verifiable?

Question 7. The first polynomial time algorithm discovered that decides PRIMES is the AKS Primality Test. When was it discovered?

A **verifier** for language A is an algorithm V such that $A = \{w : V$ accepts $\langle w, c\rangle$ for some string $c\}$. We measure the running time of a verifier as a function of the length of w.

Question 8. With respect to the definition for verifier, what does c represent? What does it within our conversation for HAMPATH? For COMPOSITES?

The Class $\mathcal{NP}$

$\mathcal{NP}$ is the set of languages that have polynomial time verifiers.
Question 9. What do the letters $\mathcal{NP}$ stand for?

Question 10. Give a polynomial time non-deterministic Turing Machine that decides HAMPATH. Show the running time of it.

The non-deterministic Turing Machine is as follows. As an aside, because n is typically used for the length of the input string, we cannot reuse it to represent the number of vertices here.

Input: $\langle G, s, t, \rangle$, where G is a directed graph with nodes s and t

1. Write a list of m numbers, $p_1, \ldots, p_m$, where m is the number of vertices in G. Each number in the list is nondeterministically selected to be between 1 and m.
2. Check for repetitions in the list. If any are found, reject.
3. Check whether $s = p_1$ and $t = p_m$. If either fail, reject.
4. For each i between 1 and $m - 1$, check whether (p_i, p_{i+1}) is an edge of G. If any are not, reject.
5. If we reach this line, all tests have passed, so accept.

Membership in $\mathcal{NP}$: A language is in $\mathcal{NP}$ iff it is decided by some nondeterministic polynomial time Turing machine.

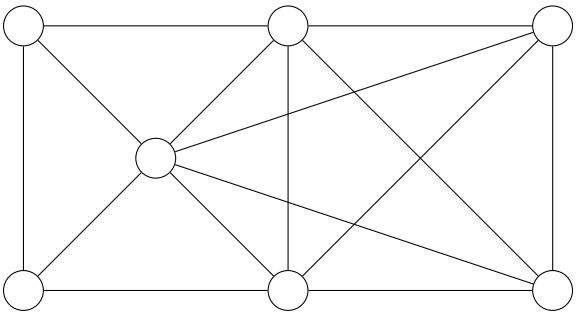
If you were to prove this formally, we would convert a polynomial time verifier to an equivalent polynomial time non-deterministic Turing Machine, and vice versa. A proof appears in the Sipser text, p.294-295.

The set $\text{NTIME}(t(n)) = \{L : L$ is a language decided by an $\mathcal{O}(t(n))$ time nondeterministic Turing Machine.$\}$

$$\mathcal{NP} = \bigcup_k \text{NTIME}(n^k)$$

Clique is in $\mathcal{NP}$

The **clique** problem is that we are given a graph G and a value k and want to decide if the given graph has a clique of size k.



Question 11. Prove this is in $\mathcal{NP}$ by providing a polynomial time verifier.

Input: $\langle\langle G, k\rangle, c\rangle$

1. Test whether c is a subgraph with k nodes in G
2. Test whether G contains all edges connecting nodes in c
3. If both pass, accept. Otherwise, reject.

Question 12. Prove this is in $\mathcal{NP}$ by providing a nondeterministic polynomial time Turing Machine to decide it.

Input: $\langle G, k\rangle$

1. Nondeterministically select a subset c of k nodes of G
2. Test whether G contains all edges connecting nodes in c
3. If yes, accept. Otherwise, reject.

Subset Sum is in $\mathcal{NP}$

The SUBSET SUM problem is that we are given a collection of numbers $x_1, \ldots, x_k$ and a target number t. We want to determine whether the collection contains a subcollection that adds up to t.

Example: $\langle\{4, 11, 16, 21, 27\}, 25\rangle \in$ SUBSET-SUM

Question 13. Prove this is in $\mathcal{NP}$ by providing a polynomial time verifier.

Question 14. Prove this is in $\mathcal{NP}$ by providing a nondeterministic polynomial time Turing Machine to decide it.

The Boolean Satisfiability Problem

SAT $= \{\langle\phi\rangle \mid \phi$ is a satisfiable Boolean formula$\}$.

A **boolean formula** is a formula of boolean variables using AND, NOT, and OR operators. A formula is satisfiable if there is an assignation of true/false values (a "TVA": a "truth value assignment") to the boolean variables such that the formula evaluates to true.

We can take a general instance of SAT and write it in the following format:

3-SAT: Given a set X of *Boolean variables* $x_1, \ldots, x_n$; each can be `true` or `false`. A *term* is either a variable or its negation. We have k clauses, each of 3 terms, disjuncted. A *truth assignment* for X is a mapping of each variable x_i to `true` or `false`. We say an assignment *satisfies* clause j if it causes that clause to evaluate to true. Our goal is to find a truth assignment that satisfies every clause.

For example, $\phi = (x_2 \vee x_3 \vee x_4)(\overline{x_2} \vee x_3 \vee \overline{x_4})(\overline{x_1} \vee x_3 \vee x_5)(\overline{x_1} \vee x_2 \vee x_5)(\overline{x_3} \vee x_4 \vee x_5)(\overline{x_2} \vee \overline{x_4} \vee \overline{x_5})$
$(x_1 \vee \overline{x_2} \vee x_5)(x_3 \vee \overline{x_4} \vee x_5)(x_1 \vee \overline{x_3} \vee \overline{x_5})(\overline{x_2} \vee x_4 \vee \overline{x_5})(x_1 \vee x_2 \vee \overline{x_4})(\overline{x_1} \vee x_2 \vee x_3)(\overline{x_1} \vee \overline{x_2} \vee x_5)(\overline{x_1} \vee x_2 \vee \overline{x_4})$

Question 15. What is a *satisfying assignment* to this?

Question 16. Prove that 3-SAT is in $\mathcal{NP}$.