Here is a problem that has a polynomial-time *randomized* solution, but so far no poly-time deterministic solution. Let F be any field and let $Q(x_1, \ldots, x_n)$ be a polynomial of degree d in n different variables. We are *not* necessarily given Q in its standard form as a sum of monomials! It may be, for example, an arithmetic product of polynomials which when computed out would be too big for us to represent. All we are guaranteed is that we can *evaluate* this polynomial given specific values of each of the variables.

Our problem is to decide whether Q is **identically zero**, meaning that it is equal to the zero polynomial as a polynomial (we write this "$Q \equiv 0$"). The problem would be trivial if Q were in standard form but consider the polynomial

$$(x_1+x_2)(x_1-x_2)+(x_2+x_3)(x_2-x_3)+(x_3+x_1)(x_3-x_1).$$

When we multiply this out, everything cancels and we get zero. But in general, the computation of the standard form of an expression of length n is not a polynomial-time operation, simply because the output might take too long to write down.

Testing for zero is really the same problem as taking *two* polynomials in this form and testing whether they are equal, since $Q \equiv Q'$ iff $Q - Q' \equiv 0$. (If we can evaluate both $Q(z_1, \ldots, z_n)$ and $Q'(z_1, \ldots, z_n)$, we can certainly evaluate their difference.) We'll also see an application of this problem to graph theory in a bit.

The main idea of **Schwartz' algorithm** (used, for example, by the Maple symbolic algebra system) is a simple one. We choose a value z_i for each variable x_i *at random*, and evaluate $Q(z_1, \ldots, z_n)$. If $Q \equiv 0$, we will always get an answer of 0. If $Q \not\equiv 0$, however, it seems reasonable that we should get a "fairly random" answer, and thus we are *likely* to get a nonzero result, which will prove that $Q \not\equiv 0$. It is possible for a *function* to be zero most of the time without being identically zero, of course, but we will show that a *polynomial* cannot do so, at least over an infinite field or a field larger than the degree of the polynomial.

One round of Schwartz' algorithm proceeds as follows:

- Choose a set S of m distinct field values.
- Choose $z_1, \ldots, z_n$ each uniformly from S, independently.
- Evaluate $Q(z_1, \ldots, z_n)$.
- If you get 0, return "true", else return "false".

As we said, if $Q \equiv 0$ we will return "true" with probability 1. We need to analyze the probability of returning "true" if Q is a nonzero polynomial of degree d – in fact we will show that this probability is d/m. Since m is under our control (as long as the field has enough elements), we can use repeated trials of the basic algorithm to reduce our error probability. If we ever get a result of "false" we know it is correct, and if we get "true" many times in a row we know that either $Q \equiv 0$ or we have been very unlucky in our random choices – we'll examine the question of how unlucky soon.

Let's first consider the case where $n = 1$, that is, Q is a polynomial of degree d in a single variable. We know something about such polynomials over any field – they can have at most d different roots. (Why? For each root r, the linear polynomial $x - r$ must divide Q. Over a field, this means that the product of the linear polynomials for each root must also divide Q, and the degree of this product is exactly the number of roots.) *It was pointed out in class that "field" is not exactly the algebraic property we need, though it is sufficient. The polynomials over a field are not a field, for example, because you cannot divide.*

For any $Q(x)$ of degree d and any S of size m, the greatest chance of getting $Q(z_1) = 0$ is if Q has d roots and all of them happen to be in S. In this case the probability is d/m, and in any other case it is less.

So we have established the bound we want in the special case $n = 1$. We will prove the d/m bound for n variables by induction on n. So we assume as inductive hypothesis that for any d and any polynomial R of degree d on $n - 1$ variables, the probability that $R(z_1, \ldots, z_{n-1}) = 0$ for random z_i's is at most d/m.

Here is a bad argument that gets at the central idea of the correct argument. Consider any sequence of values $z_2, \ldots, z_n$ each chosen from S. If we substitute these values into Q, we get a polynomial $Q(x_1, z_2, \ldots, z_n)$ in the single variable x_1, and this polynomial has degree at most d. By our argument for the one-variable case, our probability of getting $Q = 0$ is at most d/m. The probability that Schwartz' algorithm fails is the *average*, over all possible sequences $(z_2, \ldots, z_n)$, of the probability of getting 0 from that sequence. The average of these m^{n-1} numbers, that are each at most d/m, must itself be at most d/m.

Do you see the flaw in the argument? The probability of a one-variable polynomial being 0 is bounded by d/m only if that polynomial is not itself the zero polynomial! There is no reason that a particular sequence of z values might not cause every coefficient of Q to become zero, and hence cause *all* the values of z_1 to lead to $Q(z_1, \ldots, z_n)$ being zero.

So our m^{n-1} individual probabilities are *not* all bounded by d/m, as some of them may be 1. But our correct argument will put a bound on the number that are 1, in order to show that the average of all the numbers is really at most d/m.

Let's rewrite Q as a polynomial in x_1, whose coefficients are polynomials in the other $n-1$ variables:

$$Q \;=\; Q_k x_1^k + Q_{k-1} x_1^{k-1} + \ldots + Q_1 x_1 + Q_0$$

Here k is the maximum degree of x_1 in any term of Q. Note that the degree of the polynomial $Q_k(z_2, \ldots, z_n)$ must be at most $d-k$, since there are no terms in Q of degree greater than d.

How could a setting of $(z_2, \ldots, z_n)$ cause Q to become the zero polynomial? A necessary, but not sufficient condition, is that $Q_k(z_2, \ldots, z_n)$ be zero. But *by our inductive hypothesis*, we know that the probability of this event is bounded above by $(d-k)/m$, because Q_k has degree at most $d-k$ and Q_k has only $n-1$ variables.

Let's look again at our m^{n-1} individual probabilities. At most a $(d-k)/m$ fraction of them might be 1. The others are each at most d/m, as we said, but we can do better. Since the degree of Q *in the variable* x_1 is exactly k, if the values of the z_i's cause Q to become nonzero they cause it to have degree at most k. Thus the probability that the choice of z_1 causes $Q(z_1, \ldots, z_n)$ to be 0 is at most k/m.

We have the average of m^{n-1} probabilities, which is the sum of those probabilities divided by m^{n-1}. The sum of the ones, divided by m^{n-1}, is at most $(d-k)/m$. The sum of the others, divided by m^{n-1}, is at most k/m because each of them is at most $k - m$. Our total probability is thus at most d/m, completing our inductive step and thus our proof.

If we want to ensure that the probability of a wrong answer is at most ϵ, then we can set m to be $2d$ (assuming that there are enough elements in F) and run the basic Schwartz algorithm $t = \log_2(1/\epsilon)$ times. The only way we can get a wrong answer is if $Q \not\equiv 0$ and each of the trials "accidently" gave us a value of zero. Since each trial fails with probability at most $d/m = 1/2$, the probability of an overall failure is at most $2^{-t} = \epsilon$, as desired.

Remember that setting t to 1000, or even 100, drives the error probability low enough that it becomes insignificant next to other possible bad events like a large meteorite landing on the computer during the calculation.

Remember also that this **amplification of probabilities** only works when the probability of success that we start with is *nontrivial*, in particular at least an inverse polynomial $1/p(n)$. If the probability of failure is $1 - f(n)$, then the probability of t consecutive failures is $(1 - f(n))^t$ or approximately $e^{-tf(n)}$. For this to be a small probability, $tf(n)$ must be at least 1, which is to say that t must be at least $1/f(n)$.

When we study **NP**-complete problems, we will see problems that have randomized algorithms with a very small probability of success such as 2^{-n}. For example, in the SATISFIABILITY problem our input is a boolean formula in n variables and we ask whether there is any setting of the n variables that makes the formula true. It is certainly possible that only one of the 2^n settings makes the formula true, so that choosing a random setting has only a 2^{-n} probability of success. In this case t trials would have at most a $t2^{-n}$ probability of success, which is vanishingly small for any reasonable t.

This analysis depended on the fact that the basic Schwartz algorithm has only **one-sided error** – it can return "true" when the right answer is "false", but not vice-versa. Thus repeated trials could fail only if each individual trial fails, and we can multiply the individual failure probabilities to get the overall failure probability.

If we have a randomized Monte Carlo algorithm with **two-sided error**, meaning that no individual answer can be trusted, the analysis of repeated trials is different. If the answer is a bit and our failure probability is *significantly* less than $1/2$, at most $1/2 - 1/p(n)$ for some polynomial p, it turns out that taking the majority result of a sufficiently large polynomial number of repeated trials is overwhelmingly likely to be correct. If the basic failure probability is something like $1/2 - 2^{-n}$, however, a polynomial number of repeated trials don't help much.

We can apply the Schwartz algorithm to verify polynomial identities in a familiar setting – the problem of determining whether a matching exists in a bipartite graph. Given $G = (U, V, E)$, we can construct a polynomial that is identically zero iff G does *not* have a perfect matching. Thus the Schwartz algorithm can be used to either:

- Prove that the graph has a perfect matching, or
- Give us arbitrarily high confidence that it does not, given enough independent trials of the basic algorithm.

In the first case, the Schwartz algorithm will *not* give us a perfect matching in the graph, only assurance that it exists.

The polynomial we must test is the **determinant** of a particular matrix. Given any n by n matrix A, the determinant is defined to be a particular sum of products of matrix entries. For every bijection σ from the set $\{1, \ldots, n\}$ to itself (every **permutation**), we form the product of the entries $A_{i,\sigma(i)}$ for each i. Then we add these products together with coefficient 1 if σ is an **even permutation** and -1 if it is an **odd permutation**.

(What do these last two terms mean? Any permutation can be written as a product (composition) of **tranpositions**, permutations that swap two elements and keep the others fixed. The various products resulting in a given permutation are either all odd or all even, and we call the permutation odd or even accordingly. A k-cycle, which is a permutation that takes some a_1 to a_2, a_2 to a_3, etc., until it takes a_k to a_1, is odd if k is even and even if k is odd.)

So the determinant of a 2 by 2 A is $a_{11}a_{22} - a_{12}a_{21}$, and the determinant of a 3 by 3 A is

$$a_{11}a_{22}a_{33} - a_{11}a_{32}a_{23} + a_{12}a_{23}a_{31}$$
$$-a_{12}a_{21}a_{33} + a_{13}a_{32}a_{21} - a_{13}a_{31}a_{22}.$$

For general n by n this definition does not give a good way to compute the determinant because there are $n!$ different permutations to consider. But there are two good ways to compute it more quickly:

- The row operations of **Gaussian elimination** do not change the determinant, except for swapping rows which multiplies it by -1. So we can transform the original matrix to an **upper triangular** matrix by a polynomial number of these operations, counting the number of times we swap rows. The determinant of an upper triangular matrix is just the product of the entries on its main diagonal.
- There is a general method, which we won't cover here, to express the determinant of A as one entry of the product of some matrices derived from A. This means that we can in principle find the determinant in subcubic time, or find it quickly in **parallel**.

Both of these methods assume that we are carrying out the matrix operation over a field in which entries can be stored, added, and multiplied quickly. The integers, integers modulo p, real numbers, and the complex numbers are such fields, and polynomials over *one* variable (even $O(1)$ variables) still allow these operations even though they do not form a field. But as we have seen, simple operations on polynomials over many variables may produce answers that we can no longer store or manipulate easily.

Let's return to the perfect matching problem on a bipartite graph $G = (U, V, E)$, where $|U| = |V| = n$. Define the following n by n matrix A – the (i, j) entry is the variable x_{ij} if there is an edge from vertex i of U to vertex j of V, and 0 otherwise. Look at the determinant of this matrix. Any permutation σ represents a bijection from U to V. If this bijection corresponds to a matching in E, then every entry $a_{i,\sigma(i)}$ is nonzero and the product of these entries becomes a term in the determinant of A, with either a plus or minus sign.

Furthermore, if this term for σ appears in the determinant it cannot be canceled by any other term. This is because each term contains n variables, and no two contain exactly the same variables. (If $\sigma \neq \tau$, there must be some variable $x_{i,\sigma(i)}$ that is not $x_{i,\tau(i)}$.)

So given G, we test whether G has a perfect matching by repeatedly choosing random values from some set S, for each x_{ij}, getting a matrix of field elements, and evaluating the determinant of that matrix. If we ever get a nonzero value, a perfect matching exists, and if we keep getting nonzero values we build our confidence that no perfect matching exists.

How does the time of this algorithm compare with our other methods to test for perfect matchings? The network flow method takes $O(e^2n) = O(n^5)$ time as we presented it, or $O(n^3)$ by a more complicated algorithm that we did not present. Finding the determinant (as we didn't prove) takes $O(n^3)$ by the standard method of matrix multiplication or $o(n^3)$ by faster methods. So the new method is not clearly better for sequential machines, but it does allow for a fast parallel algorithm where network flow, as far as we know, doesn't.

(Note finally that the Adler notes are quoting running time in terms of n as the whole input size, not the number of vertices. So they refer to the times of my "$O(n^3)$" algorithms as "$O(n^{3/2})$".)