4 Subset-Sum (Knapsack) problems and their uses

4.1 Subset-sum problems are hard

A subset of the numbers

$$267 \quad 493 \quad 869 \quad 961 \quad 1000 \quad 1153 \quad 1246 \quad 1598 \quad 1766 \quad 1922$$

adds up to 5842. Spend a few minutes trying to find such a subset. Whether you succeed or not, I hope you are convinced the task is not trivial.

This is an example of a *subset-sum* problem. In general, we are given n natural numbers a_i and a target number T and asked to find a $S \subset \{1, \ldots, n\}$ with

$$\sum_{i \in S} a_i = T \qquad (*)$$

A seemingly simpler problem is the *subset-sum decision* problem. For a given a_i and T, decide whether there is an S for which $(*)$ holds, without being required to identify such an S. However, it can be proved that the decision problem is just as difficult as the subset-sum problem in this sense:

Theorem 17 *Suppose we had a method of solving the subset-sum decision problem. Then we could solve a subset-sum problem using the assumed method n times.*

(the n is not particularly important— the main thing is that the number of uses of the method is not too large.)

Proof: Begin by using the method to decide if T is a sum of the a_i— if not, we can stop immediately. Then use the method to determine if $(*)$ holds for some $S \subset \{2, \ldots, n\}$. If the answer is yes, we ignore a_1 for the rest of the analysis. If the answer is no, we know we must have $1 \in S$. In this second case, we continue by using the method to decide whether there is $S \subset \{3, \ldots, n\}$ with

$$\sum_{i \in S} a_i = T - a_1$$

A yes answer means we can assume $2 \notin S$, otherwise $2 \in S$.

The idea of this proof is more important than the specific result. We show that one problem is as difficult as another by showing that a method of solving

the supposedly easier problem can be used to solve another problem. This involves constructing one or several easier problems whose solution answers the hard problem. We saw another example of this idea in our discussion of the Rabin encryption system (section 3.3).

Using more elaborate versions of the techniques in Theorem 17, it can be shown that a method of solving the subset-sum decision problem could be used to solve many other problems, including:

- Factoring
- The Travelling Salesman Problem
- Any Integer Programming Problem
- The Satisfiability Problem

Don't worry if you are not familiar with the details of these problems. The important point is that they are all well-known problems for which many people have been unable to find efficient solution methods, which makes it unlikely that there is a method which solves all subset-sum decision problems efficiently (we will go into more detail on this in section 5).

The discussion above makes it plausible that some subset-sum problems are difficult. Further, there is some evidence that the "typical" subset-sum problem is not easy. V. Chvatal[3] has shown that if the a_i and T are randomly chosen, then with high probablility (i) there will be no S for which $(*)$ holds (ii) certain simple ways of proving this will not work.

4.2 A proposed public-key system based on subset-sum

As an example of an easy subset sum problem, consider the task of determining what subset of

$$1 \quad 3 \quad 6 \quad 14 \quad 27 \quad 60 \quad 150 \quad 300 \quad 650 \quad 1400 \tag{1}$$

adds up to 836. The a_i in this problem have the special property that every number is greater than the sum of all preceding numbers ($27 > 1+3+6+14$, etc). [A sequence with this property is called *super-increasing*.]

[3] "Hard Knapsack Problems," *Operations Research*, vol. 28, pp 1402–1411

1400 clearly cannot be in the set. If 650 is not in the set, we would be in trouble, since the sum of the remaining numbers is < 650, hence < 836. Thus 650 must be in the set, and we now have the task of finding numbers which add up to $836 - 650 = 186$. 300 is too big, and the same reasoning as before shows that 150 must be in the set. If we continue, it is easy to identify the desired set as $\{650, 150, 27, 6, 3\}$.

We began section 4.1 with the problem of identifying a subset of

$$267 \quad 493 \quad 869 \quad 961 \quad 1000 \quad 1153 \quad 1246 \quad 1598 \quad 1766 \quad 1922$$

which adds up to 5842. What we didn't mention before was that the a_i were carefully chosen to make them directly related to the a_i in the easy subset-sum problem (1):

$$267 \equiv 300(1000) \pmod{2039} \quad 493 \equiv 27(1000) \pmod{2039} \quad 869 \equiv 60(1000)$$

and so forth, where 2039 is a modulus chosen in advance (larger than any of the numbers in the easy subset-sum problem) and 1000 is an arbitrarily chosen number. (we must have $\gcd(2039, 1000) = 1$)

To find the subset, begin by solving $1000U \equiv 1 \pmod{2039}$, which gives $U = 1307$. If we let b_i be the numbers in the easy problem, the hard problem can be written as

$$\sum_{i \in S} (1000 b_i) \equiv 5842 \pmod{2039}$$

When we multiply by 1307, this becomes

$$\sum b_i \equiv 1307(5842) \equiv 1478$$

It is easy to identify $\{1400, 60, 14, 3, 1\}$ as a subset which adds to 1478, and the desired subset of the original system is

$$\{1246 \equiv 1400(1000) \pmod{2039}, 869 \equiv 60(1000), 1766, 961, 1000\}$$

This would seem to give us a good public-key system: a problem which is easy once some special information (the 2039 and the 1000) is known, difficult without the information. Unfortunately, the special type of subset-sum problem created in this way can be solved even without the special information. There is a sequence of papers showing how to solve special subset-sum problems and proposing a refinement which, in turn, was solved by the next paper in the sequence. This has not happened with the RSA system, but there is no guarantee that it won't!

4.3 Breaking Knapsack Cryptosystems

We will present some of the basic ideas for attacking the system described in the preceding section, and illustrate them on a small example.[4] Our source is the paper by E. Brickell and A. Odlyzko, "Cryptanalysis: a survey of recent results," in *Contemporary cryptology: the science of information integrity.*

It is not necessary to provide an efficient algorithm guaranteed to crack all instances of a cryptosystem to call its security into question, and our analysis is only an indication that the simple system of the previous section can often be broken.

Suppose we are given the sequence a_i:

$$611 \quad 929 \quad 1996 \quad 2456 \quad 2464 \quad 3594 \quad 3646 \quad 4085 \quad 5552 \quad 6765,$$

generated using an unknown multiplier V (correponding to 1000 in the previous example), an unknown modulus M, and an unknown super-increasing sequence b_i. Let $UV \equiv 1 \pmod{M}$. Then there are non-negative k_i such that $b_i = Ua_i - Mk_i$.

M must be larger than all the b_i, hence it must be significantly larger than the earliest members of the sequence.

The decryption method begins by guessing which of the known a_i correspond to a few of the early members of the sequence. A trial-and-error approach is considered feasible for practical size problems.

Suppose we have guessed that 2464, 611, and 2456 correspond to the first three b_i, and further that 2464 corresponds to b_1.

$$b_1 = 2464U - Mk_1 \qquad b_2 = 611U - Mk_2 \qquad b_3 = 2456U - Mk_3$$

U may be eliminated from pairs of equations to obtain:

$$611b_1 - 2464b_2 = M(611k_1 - 2464k_2) \qquad 2456b_1 - 2464b_3 = M(2456k_1 - 2464k_3)$$

Since M is large compared to b_1, b_2, b_3 this implies

$$611k_1 - 2464k_2 \quad \text{and} \quad 2456k_1 - 2464k_3$$

[4] Calculations in this section were done using the Calc program based on the emacs editor. Both programs are available free as part of the GNU system.

must be small— about the size of b_1, b_2, b_3 (positive or negative). In other words, $611k_1$ and $2456k_1$ must both be close to 0 or to 2464 mod 2464.

It is plausible that restrictions of this kind give a lot of information about k_1. As we look at all possible values of k_1, $611k_1$ mod 2464 is evenly distributed through the entire range from 0 to 2463. If we require, for example, that the number is either ≤ 100 or ≥ 2363, we would expect that only about 1/12 of the values for k_1 satisfy this. It seems plausible that the distribution of $2456k_1$ is independent of the distribution of $611k_1$, so a similar restriction on the second number reduces the possible values of k_1 by $(1/12)^2$.

In our example, $611k_1 \equiv s \pmod{2464}$ implies $k_1 \equiv 1355s$ and $2456k_1 \equiv 1480s$. We look at the values of s for something which makes $1480s$ mod 2464 small. Part of the results:

s	1	2	3	4	5	6	7	8	9	10	11	12
$1480s$	1480	496	1976	992	8	1488	504	1984	1000	16	1496	512

The multiples of $s = 5$ are clearly the most likely candidates. We try $s = 5$, which gives $k_1 = 1847$.

[The published algorithm recommends guessing more than 3 of the lowest a_i and determining k_i from an integer program:

$$\min W$$
$$-W \leq a_1 k_j - a_j k_1 \leq W \qquad j \geq 2$$

using a special algorithm which is polynomial-time for a fixed number of constraints.]

Since $b_1 = 2464U - 1847M$ implies U/M is close to 1847/2464, we try using $U = 1847$, $M = 2464$ on the a_i, which gives:

a_i	611	929	1996	2456	2464	3594	3646	4085	5552	6765
$1847a_i$	5	919	468	8	0	102	50	227	1840	11

This is *not* a super-increasing sequence, since $5 + 8 > 11$. However, it is close enough to one that one can use it to solve subset-sum problems with a small amount of work. (in this example, considering separately the cases in which the a_i corresponding to 8 is and is not in the set)

The a_i were generated using $M = 6789$, $V = 1234$, $U = 5089$. Comparison of the $a_i \equiv Vb_i \pmod{M}$ with the "almost" super-increasing sequence

obtained above shows that, except for the first few terms, the ratio of terms from the two sequences is nearly constant. Since a multiple of a super-increasing sequence is super-increasing, this explains why we would expect to obtain a useful sequence.

a_i	2464	611	2456	6765	3646	3594	4085	1996	929	5552
b_i	13	17	35	66	157	300	647	1300	2537	5099
$1847a_i$	0	5	8	11	50	102	227	468	919	1840

4.4 Other uses of the subset-sum problem

The results mentioned at the end of the last section do *not* contradict the presumed difficulty of subset-sum problems in general. It is only the specially constructed problems which are known to be easy. There are security problems other than public-key codes for which subset-sum problems are useful.

4.4.1 Computer passwords

A computer needs to verify a user's identity before allowing him or her access to an account. The simplest system would have the machine keep a copy of the password in an internal file, and compare it with what the user types. A drawback is that anyone who sees the internal file could later impersonate the user.

I believe this alternative is actually implemented on some systems: the computer generates a large number (say 500) of a_i. They are stored in the internal file. A password is a subset of $\{1, \ldots, 500\}$. (in practice, there is a program to convert a normal sequence-of-symbols password to such a subset.) Instead of having the password for the user, the computer keeps the total associated with the appropriate subset. When the user types in the subset, the computer tests whether the total is correct. It does not keep a record of the subset. Thus impersonation is possible only if somebody can reconstruct the subset knowing the a_i and the total.

4.4.2 Message verification

A sender (S) wants to send messages to a receiver (R). Keeping the message secret is not important. However, R wants to be sure that the message he is

receiving is not from an imposter and has not been tampered with. S and R agree on a set of a_i (say 500) and a set of totals T_j (say 200). These numbers may be publicly known, but only S knows which subsets of the a_i correspond to which T_j. The message sent by S is a subset of size 100 of $\{1, \ldots, 200\}$. He does this by sending 100 subsets of the a_i corresponding to the message he wants to send.

5 Subset-Sum Problems and NP-Completeness

The phrase "NP-complete" has an intimidating sound. In this section, we will first define a new problem involving formulas in logic, called the Satisfiability Problem (SP). We will use the abbreviation (SSP) for the subset-sum problem. Our main results will be:

1. If there is an algorithm which efficiently solves (SSP), it can be used to efficiently solve (SP).
2. If there is an algorithm which efficiently solves (SP), it can be used to solve (SSP).
3. An algorithm to solve (SP) efficiently would give efficient solutions to factoring and many other problems.

5.1 The Satisfiability Problem

We will use capital letters A_i, B_i, to stand for logical variables. These stand for statements like "221 is a prime number" or "`TH` is the most common two-letter sequence in English," which are either true or false, i. e., these variables have values of either **T** or **F**. $\sim A_i$ ("not A_i") is the statement that A_i is false, so it has value **T** if A_i has value **F**, and $\sim A_i$ has value **F** if A_i has value **T**. We will also be interested in more elaborate formulas:

$$A_1 \text{ or } \sim A_2 \text{ or } \sim A_4 \text{ or } A_7 \text{ or } A_8$$

This formula says that either A_1 is true or A_2 is false, or A_4 is false, etc. The value of this formula will be **T** unless $A_1 = \mathbf{F}$, $A_2 = \mathbf{T}$, $A_4 = \mathbf{T}$, $A_7 = \mathbf{F}$, $A_8 = \mathbf{F}$. Thus, there is only one way in which the formula will be false.

Figure 1 illustrates a satisfiability problem. We want to assign **T**, **F** to all the variables so that all of the formulas have value **T**. Even in this small example, it may take you a minute or so to find such an assignment.

$$
\begin{array}{c}
A_1 \text{ or } A_2 \\
\sim A_1 \text{ or } A_5 \\
\sim A_1 \text{ or } A_3 \text{ or } A_4 \\
A_3 \text{ or } \sim A_5 \\
A_4 \text{ or } A_5 \\
\sim A_3 \text{ or } \sim A_4 \\
\sim A_2 \text{ or } A_3
\end{array}
$$

Figure 1: A small (SP)

5.2 Converting (SP) to (SSP)

We want to construct numbers a_i and a target number T so that there is a subset adding up to T if and only if there is an assignment for (SP) which makes all the formulas true. Using this construction allows us to use an algorithm for (SSP) to solve (SP). It implies that solving (SSP) is at least as hard as solving (SP).

We will illustrate the construction using the example in Figure 1. We will have numbers $a_1, \ldots, a_5$ corresponding to the logic variables $A_1, \ldots, A_5$ with $A_i = \mathbf{T}$ corresponding to a_i being included in the subset. We will also need additional a_i, $i > 5$ for technical reasons.

$$
\begin{array}{rr|c|c|c|c|c|c|c}
a_1 = & 1 & 01 & 01 & 00 & 00 & 00 & 00 \\
a_2 = & 2 & 00 & 00 & 00 & 00 & 00 & 01 \\
a_3 = & & & 2 & 01 & 00 & 01 & 02 \\
a_4 = & & & 4 & 00 & 01 & 02 & 00 \\
a_5 = & & 2 & 00 & 02 & 02 & 00 & 00 \\
a_6 = & 1 & 00 & 00 & 00 & 00 & 00 & 00 \\
a_7 = & 2 & 00 & 00 & 00 & 00 & 00 & 00 \\
a_8 = & & 1 & 00 & 00 & 00 & 00 & 00 \\
a_9 = & & 1 & 00 & 00 & 00 & 00 & 00 \\
a_{10} = & & 4 & 00 & 00 & 00 & 00 & 00 \\
a_{11} = & & & 1 & 00 & 00 & 00 & 00 \\
a_{12} = & & & 2 & 00 & 00 & 00 & 00 \\
a_{13} = & & & 3 & 00 & 00 & 00 & 00 \\
a_{14} = & & & 8 & 00 & 00 & 00 & 00 \\
a_{15} = & & & & 1 & 00 & 00 & 00 \\
a_{16} = & & & & 3 & 00 & 00 & 00 \\
\ldots = & & & & & \ldots & & \\
T = & 4 & 04 & 08 & 04 & 04 & 04 & 04
\end{array}
$$

Figure 2: Subset-sum problem

The subset-sum problem is shown in Figure 2. For clarity, we have divided the numbers into zones. T will be a sum of a subset of the a_i if and only if the totals within each zone are appropriate. Each zone corresponds to one of the logic formulas. For a_1 through a_5, the value in a zone is 0 if the corresponding A_i does not appear in the logic formula. If A_i does appear, a power of 2 is used. (the reason for using powers of 2 is that different subsets of $\{a_1, \ldots, a_5\}$ will have different totals)

The leftmost zone corresponds to the first logic formula in our example: A_1 or A_2. By making suitable decisions about inclusion of a_6 or a_7, we will be able to get a total of 4 in this zone, unless both a_1 and a_2 are left out of the set, which is precisely what would make the logic formula have value **F**.

The second zone corresponds to $\sim A_1$ or A_5, which has value **T** unless a_1 is in the set and a_5 is not. a_8, a_9, and a_{10} can be used to get the total for the zone equal to 4 in any other case.

Similarly, each of the other zones[5] has a_i associated with it which can be used to obtain the correct total except in one case.

The general problem is that we want numbers which can be used to obtain any total between 1 and 2^n, except for one "forbidden total" M. [In the two zones discussed above M is 4 in one case and 3 in the other] We start with the powers of 2 from 1 to 2^n. If $2^j \leq M < 2^{j+1}$, replace 2^j by the two numbers $M - 2^j$ and $M + 1$. [We did not follow exactly the procedure described in this paragraph in constructing Figure 2.]

5.3 Converting (SSP) to (SP)

Suppose we have a subset-sum problem with 50 a_i, all between 1 and 2^{20}, with $T < 50(2^{20}) < 2^{26}$. Solving the SSP may be crudely broken into two steps:

1. Decide which a_i are in the subset.

2. Verify that the sum of the chosen a_i is T.

Our (SP) will also carry out these steps. The first is simple: we will have logic variables $A_1, \ldots, A_{50}$ with $A_i = \mathbf{T}$ corresponding to a_i being in the

[5] We have omitted the a_i for the last three zones in Figure 2.

subset. To carry out the second step, we need to construct a set of logic formulas which acts as an "adding machine" to check the total.

We will represent numbers in base 2. Since all relevant numbers are $< 2^{26}$, numbers may be represented by 26 logical variables. For each $1 \leq i \leq 50$, we will have variables $B_{i1}, \ldots, B_{i26}$. These will represent a_i if $A_i = \mathbf{T}$, 0 if $A_i = \mathbf{F}$. To do this, we need formulas which show how the value of A_i determines the value of all the B_{ij}:

$$\sim A_i \text{ or } B_{ij} \quad A_i \text{ or } \sim B_{ij} \quad \text{if } j\text{th digit of } a_i = 1$$

with the simple formula $\sim B_{ij}$ if the jth digit of a_i is 0.

Next, we need, for $2 \leq i \leq 50$, variables $C_{i1}, \ldots C_{i26}$ which represent the total of the first i of the numbers given by the B-variables. Formulas are needed which show the B-variables determining the C-variables.

We begin with a set of formulas $\mathcal{S}(V, W, X, Y, Z)$ which have Y get value $\mathbf{T}$ if and only if an odd number of V, W, X have value $\mathbf{T}$. Z gets value $\mathbf{T}$ if and only if 2 or 3 of V, W, X have value $\mathbf{T}$:

$$\begin{array}{ll}
V \text{ or } W \text{ or } X \text{ or } \sim Y & \sim V \text{ or } \sim W \text{ or } Z \\
\sim V \text{ or } W \text{ or } X \text{ or } Y & \sim V \text{ or } \sim X \text{ or } Z \\
V \text{ or } \sim W \text{ or } X \text{ or } Y & \sim W \text{ or } \sim X \text{ or } Z \\
V \text{ or } W \text{ or } \sim X \text{ or } Y & V \text{ or } W \text{ or } \sim Z \\
\sim V \text{ or } \sim W \text{ or } X \text{ or } \sim Y & V \text{ or } X \text{ or } \sim Z \\
\sim V \text{ or } W \text{ or } \sim X \text{ or } \sim Y & W \text{ or } X \text{ or } \sim Z \\
V \text{ or } \sim W \text{ or } \sim X \text{ or } \sim Y & \\
\sim V \text{ or } \sim W \text{ or } \sim X \text{ or } Y &
\end{array}$$

If V, W, X represent three one-digit numbers (0 or 1), the formulas $\mathcal{S}(V, W, X, Y, Z)$ have the effect that Y is the number in the column with the three numbers, while Z shows whether there is a number carried into the next column.

We will use $D_{i1}, \ldots, D_{i27}$, $2 \leq i \leq 50$, to keep track of numbers being carried. Since there are no numbers carried in the rightmost column, we have the formulas $\sim D_{i1}$. The formulas

$$\mathcal{S}(B_{1j}, B_{2j}, D_{2j}, C_{2j}, D_{2(j+1)}) \qquad 1 \leq j \leq 26$$

have the effect of making C_{2j} represent the sum of the numbers B_{1j} and B_{2j}. To have C_{ij} represent the sum of $C_{(i-1)j}$ and B_{ij}, we use

$$\mathcal{S}(B_{ij}, C_{(i-1)j}, D_{ij}, C_{ij}, D_{i(j+1)}) \qquad 3 \leq i \leq 50 \quad 1 \leq j \leq 26$$

These logic formulas together have the effect that the A_i determine the B_i, which determine C_{2j} through C_{50j} successively. This last group of variables corresponds to the base-2 representation of the sum of the a_i which are in the set we have chosen. Finally, we add the formulas C_{50j} or $\sim C_{50j}$ depending on whether the jth digit of T is 1 or 0. As planned, a solution to this satisfiability problem gives a solution to the subset-sum problem (look at which A_i have value **T**), which implies that a method of solving (SP) can be used to solve (SSP).

The (SP) we have constructed is rather large, involving approximately 15(26)(50) formulas. However, the rate of growth if we had more a_i with a larger upper limit is not too bad.

5.4 Cook's Theorem

It is more important to understand the general idea of what we did in section 5.3 than to get involved in the details of the construction of the "adding machine." We used logical variables (the A_i) to represent our guess as to what a solution to the (SSP) was, then constructed a set of formulas to verify the guess was correct.

You should be able to convince yourself that a similar thing could be done with a factoring problem. We could have variables A_i and B_i represent our guesses as to two factors, then construct a "multiplication machine" to verify that the product is what we want. Thus an efficient algorithm for (SP) would lead to an efficient algorithm for factoring[6].

Many other problems can be viewed as making some guesses as to what the correct answer is, with the process of verification relatively easy. An example might be an integer programming problem:

$$\begin{aligned} &\max cx \\ &Ax \leq b \\ &x_i = 0 \text{ or } 1 \end{aligned}$$

We ask if there is a feasible solution with objective function value $> K$. For a given x, it is easy to check that it satisfies all the problem constraints and tell if the value is big enough.

[6]Unlike (SSP), nobody has been able to show that an algorithm for factoring would give an algorithm for (SP).

The detailed construction in section 5.3 was intended primarily to convince you that, if a verification can be done efficiently, it can be simulated by a set of logic formulas. It should make you willing to believe

Theorem 18 (Cook) *Any "guess and verify" problem can be converted to a satisfiability problem. Thus, an efficient algorithm for (SP) can efficiently solve any "guess and verify" problem.*

5.5 Terminology

The concept we have vaguely described as solving efficiently is technically known as "polynomial time." The types of problem that can be considered as "guess and verify" are called NP (for Nondeterministic [the guessing stage] Polynomial [the verification stage]). Cook's Theorem says that (SP) is as hard as any NP problem— the usual terminology is to say (SP) is NP-complete. Since we showed in section 5.2 that (SSP) could be used to solve (SP), we essentially proved that (SSP) is also NP-complete.

Computers and Intractability, by Garey and Johnson, is strongly recommended for more information.

6 A probabilistic test for primality

Suppose we want to test whether 247 is a prime number. Recall two facts about prime numbers p:

1. $a^{p-1} \equiv 1 \pmod{p}$ if $a \not\equiv 0$.
2. If $a^2 \equiv 1 \pmod{p}$, then $a \equiv 1$ or $a \equiv -1$. $[(a+1)(a-1) \equiv 0]$

Suppose we randomly choose $a = 2$ and test for consistency with these conditions. Since $2^{246} \equiv 220 \pmod{247}$ we can conclude immediately that 247 is *not* a prime.

Perhaps we were lucky with $a = 2$. If we try $a = 27$, we get $a^{246} \equiv 1 \pmod{247}$. However,

$$27^{246} \equiv \left(27^{123}\right)^2 \text{ and } 27^{123} \equiv 170 \pmod{247}$$

which is inconsistent with the second condition, again implying 247 is not a prime.

Not every choice of a is inconsistent with the conditions. For example, $160^{123} \equiv -1$ (hence $160^{246} \equiv 1$) and $178^{123} \equiv 1$. However, the fact that some choices of a give a proof that a number is not prime suggests the following test:

Rabin's Primality Test. Let $p-1 = 2^k m$, where m is an odd number. Choose a at random. Compute the sequence

$$a^m \quad a^{2m} \quad a^{4m} \ldots a^{p-1} \qquad \mod p$$

This sequence is consistent with p being a prime if $a^m \equiv 1$ or if the sequence has -1 at some point, followed by 1 for all subsequent terms. In all other cases, a provides a proof that p is not prime (this is usually described by saying that a is a *witness* that p is not prime). Repeat this test for some number of random choices of a, and conclude that p is a prime if none of the chosen a is a witness.

Two features of the test should be emphasized. It does not provide an absolute guarantee that p is a prime, only that it is probably a prime (we will analyze exactly how probable in the next section). Secondly, when we know p is not a prime, we do not know what its factors are— factoring is much more difficult than testing for primality.

[A different probabilistic test is described near the end of the RSA paper.]

6.1 Analysis of the Rabin test

We will calculate how many a are witnesses that 247 is not a prime. Our analysis will make use of the fact that $247 = 13(19)$ and that 2 is a primitive root for both 13 and 19. However, it should be emphasized that this information (which will not be available in general) was not used when we did the test itself.

How many a satisfy $a^{123} \equiv 1 \pmod{247}$? We must have $a^{123} \equiv 1 \pmod{13}$ and $a^{123} \equiv 1 \pmod{19}$. Let $a \equiv 2^x \pmod{13}$. Then we must have $123x$ divisible by 12. This gives the possible values $0, 4, 8$ for x, which implies $a \equiv 1, 3$, or 9 mod 13. Similarly, if $a \equiv 2^x \pmod{19}$, $123x$ must be divisible by 18, which leads to $a \equiv 1, 7$, or 11 mod 19. (we actually found 178 above by solving $a \equiv 9 \pmod{13}$ and $a \equiv 7 \pmod{19}$) The 3 choices mod 13 and mod 19 imply there are 9 a with $a^{123} \equiv 1$.

How many a satisfy $a^{123} \equiv -1 \pmod{247}$? If $2^{123x} \equiv -1 \pmod{13}$, we must have $123x \equiv 6 \pmod{12}$, which leads to $a \equiv 4, 7$, or 17 mod 13. Similarly, we get $a \equiv 8, 18$, or 12 mod 19. Thus we get 9 a satisfying this condition.

If we choose $1 \leq a \leq 246$ at random, the chances of getting an a that is not a witness are $18/246 \approx .073$. If we do the test 5 times, the chance of incorrectly concluding 247 is a prime is $\approx 2(10^{-6})$.

[We actually did more work than necessary, identifying the exact set of numbers which would lead to a wrong conclusion. If we only want to count how many numbers there are, we could make use of observations such as that, for any k, an equation $123x \equiv k \pmod{12}$ will either have 3 solutions or no solutions.]

Theorem 19 (Rabin) *If p is not a prime, at least $3/4$ of $1 \leq a \leq p-1$ are witnesses.*

This implies that for any non-prime p, the chance of being incorrectly identified after 5 tests is $\leq 4^{-5} < .001$.