Automated Cryptogram Solving, Volume IV, Appendices

Design Guide to Automatically Solve Almost All of the ACA Cipher Systems by Computer

By SHMOO

Contains: Sample Subroutines, Supplemental Topics, Useful Statistics

Appendix A. Sample Subroutines and Classes

The point of this appendix is to provide short programs that illustrate some key concepts, especially those useful in many programs. By themselves, none of these solve ciphers, but they do provide some critical "leverage" for the programmer so they can code up efficient solutions in their own favorite language. Python was selected because it expresses ideas directly and compactly. The amount of "boilerplate" is low. As a bonus, many members either know the language or can work it out by studying the examples.

Factorial Key Set

This is a class (that is, an object) which houses a "factorial." In several cases, especially in exhaustive solutions, the coder will want to iterate and collect all possible N factorial keys. This object can be used iteratively to access successive arrays that have 'the next' factorial set in it and so obtain them all over time. Much faster than 10 to the Nth power with eliminations when N is the period size.

```
class FactorialKeySet :
    #
    # by SHMOO
    #
    # FactorialKeySet returns all possible keys ("zero origin") for
    #    a given integer.  So FactorialKeySet(5) returns all 5! values, iteratively,
    #    if it is invoked correctly.
    #    Note that values from 3 to 12 are supported.
    #    Keys are not returned in any particular order, but all N! are returned.
    #    Note that this is a Python class (aka "object").
    #
    #    Typical invocation is:
    #
    #    factS = FactorialKeySet(5)  # 5! keys to process here.
    #    key = factS.start()  # required. Makes first key,copies to key
    #    while (factS.hasMore()) : # Have all keys been tried?
    #       pt = trialDecipher(key)
    #       score = grade(pt)
    #       if(score>bestScore) :
    #           saveCurrentKeyAndAnswer(key,pt,score)
    #           bestScore=score
    #       key = factS.next()
    #   # done here.
    #
    #   Note that for coherence and to terminate you _must_ execute
    #   a new key = factS.next()
    #   in each "while" iteration unless you abandon the loop altogether.  Each
    #   "next" consumes a key so use it accordingly.  "start" does also.
    #
    unitTest=False  # Turn this on to execute the unit test.
    fourFactorial = [  # These arrays optimize execution
        [ 3, 2, 1, 0 ],        [ 2, 3, 1, 0 ],
        [ 2, 1, 3, 0 ],        [ 2, 1, 0, 3 ],
        [ 3, 1, 2, 0 ],        [ 1, 3, 2, 0 ],
        [ 1, 2, 3, 0 ],        [ 1, 2, 0, 3 ],
        [ 3, 1, 0, 2 ],        [ 1, 3, 0, 2 ],
        [ 1, 0, 3, 2 ],        [ 1, 0, 2, 3 ],
        [ 3, 2, 0, 1 ],        [ 2, 3, 0, 1 ],
        [ 2, 0, 3, 1 ],        [ 2, 0, 1, 3 ],
        [ 3, 0, 2, 1 ],        [ 0, 3, 2, 1 ],
        [ 0, 2, 3, 1 ],        [ 0, 2, 1, 3 ],
        [ 3, 0, 1, 2 ],        [ 0, 3, 1, 2 ],
        [ 0, 1, 3, 2 ],        [ 0, 1, 2, 3 ],
        [ 17, 17, 17, 17 ] ] # simplifies implementation do not return
    threeFactorial = [
        [ 2, 1, 0 ],        [ 1, 2, 0 ],        [ 1, 0, 2 ],        [ 2, 0, 1 ],
```

```
        [ 0, 2, 1 ],        [ 0, 1, 2 ],        [ 17, 17, 17 ]]
    twoFactorial = [
        [ 1, 0],
        [ 0, 1] ,[ 17,17]]
    oneFactorial = [ [0], [17] ]
    def calcFact(x) :
        if(x==1) : return 1
        return x*FactorialKeySet.calcFact(x-1)
    def __init__(this,factorial) :
        this.fact = int(factorial)
        if(this.fact<1 or this.fact>12) :
            raise ValueError("Factorial Key Set unsupported value "+str(value))
        this.sizeFact= FactorialKeySet.calcFact(factorial)
        this.currLoc=0
        if(factorial==1) :
            this.fArray=FactorialKeySet.oneFactorial
        elif(factorial==2) :
            this.fArray=FactorialKeySet.twoFactorial
        elif(factorial==3) :
            this.fArray=FactorialKeySet.threeFactorial
        elif(factorial==4) :
            this.fArray=FactorialKeySet.fourFactorial
        else :
            this.nextA = FactorialKeySet(factorial-1)
    def start(this) :
        if(this.fact<5) :
            this.currLoc=this.currLoc+1
            return this.fArray[0]
        this.downArray= this.nextA.start()
        this.currArray=[]
        this.currArray.append(this.fact-1)
        this.currArray.extend(this.downArray)
        this.currLoc=this.currLoc+1
        this.nextInsert=1
        return this.currArray
    def next(this) :
        if(this.currLoc>this.sizeFact) :
            raise ValueError("Factorial Key Set returns too many elements
"+str(this.currLoc))
        if(this.fact<5) :
            this.currLoc=this.currLoc+1
            return this.fArray[this.currLoc-1]
        if(this.nextInsert>=this.fact) :
            this.downArray=this.nextA.next()
            this.currArray=[]
            this.currArray.append(this.fact-1)
            this.currArray.extend(this.downArray)
            this.currLoc=this.currLoc+1
            this.nextInsert=1
            return this.currArray
        j=0
        for i in range(0,this.fact) :
            if(i!=this.nextInsert) :
                this.currArray[i]= this.downArray[j]
                j=j+1
            else :
                this.currArray[i]= this.fact-1
        this.nextInsert = this.nextInsert+1
        this.currLoc=this.currLoc+1
        return this.currArray
    def hasMore(this) :
        if(this.currLoc<=this.sizeFact) :
            return True
        else :
            return False
```

Factors (Dimensions) for Arrays, From Text size 2 to 100

In ACA puzzles, most transposition ciphers are one hundred characters or less. The text may decompose into a rectangle that is a pair of factors that multiply to the ciphertext length. This is particularly important in Route Transpositions. There are many ways to deal with these facts. The following is a short bit of code that returns an "array of arrays" that show the non-trivial factors in ascending order.

```
# Calculate factors for (e.g.) route transpositions by
#   length.  Can either give them for a specific length
#   or print all out from (say) 3 to 100 as precomputed
#   values.  The number of factors will vary and if 30
#   is, in part 3x10 it isn't necessary to return 10x3
#
def FindFactors(ctLen) :
    res=[]
    for i in range(2,ctLen//2) : # Integer division
        val = ctLen//i # integer division, so truncates
        if(ctLen==(val*i)) : # divides evenly only if equal
            if(val>=i ) : # already recorded it or prime
                pair=[]
                pair.append(i)
                pair.append(val)
                res.append(pair)
    return res
#
for leng in range(20,28) : #demonstration portion, us 2 to 100 for full set
    res = FindFactors(leng)
    print("For "+str(leng)+" factors are "+str(res))
```

Execution looks like this:

```
For 20 factors are [[2, 10], [4, 5]]
For 21 factors are [[3, 7]]
For 22 factors are [[2, 11]]
For 23 factors are []
For 24 factors are [[2, 12], [3, 8], [4, 6]]
For 25 factors are [[5, 5]]
For 26 factors are [[2, 13]]
For 27 factors are [[3, 9]]
```

. . .but note that the FindFactors' output can be managed many ways; this is for illustrative purposes.

Appendix B. Supplemental Topics

Here are some "deep dive" details that might interest those writing code.

Topic One: Ordered Lists and the Languages that Provide Them

A recurrent theme in cryptographic programming is an Ordered list. There is something – a tetragram, a score, that is not continuous but unique. It can form a key. Associated with the key is a value – a frequency, a trial decipherment, something we value. This is something that every cryptographer needs in their bag of tricks. If you implement hill climbing, you want to collect, ordered by highest score, the best solutions and print them from highest to least score at the end. Not just the very highest score – the five, ten, fifty, two hundred top scores (how many varies by the system). This covers a host of sins, including specific ciphertexts that are not perfectly vulnerable to a given program, but none the less the right answer is visible if one looks through an ordered list.

If you are calculating periods, you want to try each plausible period out and then collect the results sorted by highest to lowest score and then process only the plausible periods but often more than one.

There are other examples, but this should be motivation enough. Ordered lists are simply a recurrent theme.

Different languages call it different things. Some sort as they go, others require you to sort the keys at the end. Python call them "Dictionaries". Java calls them TreeMaps. Whatever their name, learn how to use them. What if your language (for instance, C) does not have this? Implement your own. It is not a small piece of code, but it will more than repay your efforts because it simplifies your coding for every system. (A really enterprising C coder might be able to encapsulate a small bit of C++ 11 code in a wrapper that C can call).

Sooner or later, you want your autosolvers to run in batch mode. That is, you want to launch your solver for Patristocrats against every "Pat" in the issue while you go make a sandwich. You will discover an ordered list of solutions becomes a requirement.

A short Python example will show the value and how it works in that language. Python Dictionary objects have unique keys, but not pre-sorted. However, the "sorted" verb will easily create the sorted/ordered keys whenever you need them.

```
import io
rawData = [ # raw, English monographic frequencies from some source
25245175,"A",   4587494,"B",    9971582,"C",    11579824,"D",
35858591,"E",   6527515,"F",    5828761,"G",    14035236,"H",
22031592,"I",   587921,"J",     1962700,"K",    12429418,"L",
7701600,"M",    21447961,"N",   21400624,"O",   6291322,"P",
314724,"Q",     19357123,"R",   19910045,"S",   25243749,"T",
7774153,"U",    3178703,"V",    4853333,"W",    619111,"X",
4610012,"Y",    379235,"Z"]
dict = {}  # our Python dictionary object
for i in range(0,len(rawData)-1,2) :
    dict[rawData[i]]=rawData[i+1]
keyList = sorted(dict, reverse=True)  # sorts keys of "dict" in
reverse order
for k in keyList :
    print("key: "+str(k)+" value: "+dict[k])
```

Execution gives (first five lines only):

key: 35858591 value: E
key: 25245175 value: A
key: 25243749 value: T
key: 22031592 value: I
key: 21447961 value: N

As can be seen, this “dictionary” of raw English monographic frequencies is reverse ordered.

Here’s a list, by language, of what objects they have to assist in this:

Language	Unique Key object available	Unique Key Presorted?
C	Nothing, write your own.	Up to you
C++	“map” which presorts keys starting in C++ 11.	Add a parameter to constructor to get descending order
Java	TreeMap	Yes, can traverse in descending order.
Python	“dictionary” is unique, not presorted	Use built-in “sorted” verb to get keys in descending order.
Pascal/Delphi	TDictionary	Use TArray.sort in reverse order
Free Pascal	TFPGMap may work	Not clear how to sort
C# / .NET / Visual Basic	SortedList	Access Sorted list in reverse index order
Javascript	Dictionary available	Can create sorted key list

Note that all of them, pre-sorted or not, share a property: The key is unique. If you say: dict[x]=3 and “x” already is an index item, the previous value becomes a 3. That often is not what we want, especially for the results of hill climbs where x is a score that might be the same for different trial plaintexts. Here’s a trick that gets around that:

```
# Illustrative code to score arbitrary hill climb scores in an
#   ordered list.  Non-unique scores are preserved as long as
#   "shifter" is well-chosen.
#
# "filter" is a limiting function for illustration.
#   In a real example, a variety of "filter" functions
#   can be devised to eliminate low scoring, false answers.
#   80 per cent of the current, known top score often works well.
#   This often allows "shifter" to be a known, empirical value.
#
def filter(score,topScore) :
    if(score> (topScore//2)) :
        return True
    return False
# pts and scores are illustrative, fake results of three hill climbs.
# The first and third plaintexts are nearly alike but score the same.
pts = ["blahblahblah", "asdrexcge", "blahblaxblax"]
scores= [  30, 20, 30] # score for each plaintext pts
shifter = 1000   # Must be greater than the number
```

```
                # of (e.g.)  successful hill climbs
sols = {}
topKnownScore = 38
scoreInstance=0
for i in range(0,3) :
    if(filter(scores[i],topKnownScore)) :
        scoreInstance=scoreInstance+1  # Unique for each
                                       # qualifying climb.
        # keyForScore preserves all qualified scores, unique or not.
        keyForScore = (scores[i]*shifter)+scoreInstance
        sols[keyForScore]=pts[i]
keyList2 = sorted(sols, reverse=True)
for k in keyList2 :
    print("instance Score: "+str(k)+" value: "+sols[k])
```

By having the actual score shifted by a safe amount and an instance value added, duplicate scores (and their maybe varying solutions) can be kept. Here is what it looks like, which can look like actual trial solutions for a real hill climber with multiple climbs:

```
instance Score: 30003 value: blahblaxblax
instance Score: 30001 value: blahblahblah
instance Score: 20002 value: asdrexcge
```

As can be seen, the correct answer is the second score. The top score is not always the answer, so this kind of function, besides enabling batch operation, can be a very large difference maker in the overall effectiveness of the code.

A Short Word About Unordered Lists

Unordered lists are also very useful. This is especially true for "sparse" lists where the key values are not nearly enough alike. Word lists, for instance, may work well as an unordered list; a lot of code doesn't necessarily care about alphabetical order, just about getting "the next one" or "this specific one".

It is tempting to implement things like tetragram lists with these, but until you get to 5 or 6-grams, which for ACA purposes hasn't worked well, an ordinary array (even of double precision values) usually performs better.

Topic Two: "Random" numbers

Generating good pseudo-random numbers is surprisingly tricky. Experience suggests that better quality randomness will give better results with otherwise identical code. Laypersons who don't want to read thick, math-laden tomes should find out the best practices and use those as opposed to settling for random "looking" numbers.

Computer Languages and Random Number Libraries

Earlier in computer history, especially when space and processor cycles were precious, a lot of compromised "random" number generators were written. Most infamously, the standard C version of it from random.h was and is not good. For compatibility, some of these compromised generators are still around. Momentarily, we will show recommended "best practices" as of 2023 for popular languages. Use library code rather than your own if possible. Subtle bugs abound that ruin randomness.

The best practice in actual coding for getting random numbers is to use whatever the language and interface has for “randomNextDouble()”. To make an integer, you simply multiply the double by the relevant value and truncate to integer:

```
integer nextRandToUse = TruncateToInteger(26* randomNextDouble());
```

This gets values between 0 and 25. The reason for getting integers in this manner is that for many generators, the higher bits are “more random” than the lower bits of the number.

Of course, nobody wants to do a graduate level study of the topic. So, how exactly are the best practice random number generators accessed, by language choice?

Here’s a useful table:

Language	**Library to Use / Code Snippet**	**Estimated Quality**
C	Consider invoking C++’s MT19937. If it must be straight C, use rand48().	Don’t use random.h. MT19937 (Mersenne Twister) is better. rand48() passes “dieharder” suite reasonably well.

C++	#include <random>
	std::random_device rd{};
	std::mt19937_64 engine{rd()}; // Mersenne Twister
	std::uniform_real_distribution<double>
	dist{0.0,0.999999999};
	
	double nextR = dist(engine)
		Uses Mersenne Twister, which seems well regarded and passes “dieharder” suite very well.

Java	import java.util.concurrent.ThreadLocalRandom; Random rand = ThreadLocalRandom.current(); Double nextR = rand.nextDouble();	Uses SplitMix. Tests well in “dieharder”. Thread safe.
Python	import random nextR = random.random()	Uses Mersenne Twister
Pascal / Delphi	Consider invoking C++’s MT19937 somehow	Quality unclear, wrapper C++ if possible
Free Pascal	nextLI = Random()	Mersenne Twister, “presently”, generates 64 bit integer
C# / .NET / Visual Basic	Random rand = new Random() nextR = rand.nextDouble();	Variation of a Knuth algorithm. Very obscure source, but tests well in dieharder.
Javascript	let nextR = Math.random()	Implementation defined – be wary!

An alternative to all of this is Cryptographically Random Number Generators. Deploying even these can be tricky; one should always test as implementation trifles can lead to a bad result. The best non-crypto ones are going to be considerably faster; an issue as they are invoked in the "deep loops" of the code. They are also easier to understand and, thus, deploy correctly.

Really, Truly Random Numbers

It is possible, albeit somewhat slowly, to obtain genuine random values from real world, physical sources. This is not recommended for the likes of us – it take care and measurement. See https://www.random.org/ for one organization that is working on this problem.

Topic Three: Recursion

The most general programming technique of all is recursion. The classic example is Factorial. It is defined to 'call itself' to get the answer.

In Python:

```
import io
def factorial(x) :
    if(x<=1) : return 1
    return x*factorial(x-1)
for i in range(3,8) :
    print(i,factorial(i))
```

```
Execution:
3 6
4 24
5 120
6 720
7 5040
```

Recursion is difficult for some programmers, but it comes up as the "right technique" over and over again in cryptography applications.

A key trick: It is usually a very good idea to remember "the last shall be first." Meaning, the first thing to do in a recursive routine is to check for the end condition. You see that in this basic example with if(x<=1) : return 1 . . . this is a classic bit of code in a classic example.

Another thing that may have to be done is to implement an escape. That is, throw an exception, based on total recursive calls of the function, and catch it at the top level, dealing with incomplete results.

Recursion has a lot in common with the next topic, Multi-threading. Static values that change must be done carefully in recursion, even if single-threaded.

Topic Four: Multi-threading

There are two completely different ways to look at this question:

1. Cryptography is an ideal problem for multi-threading
2. For ACA systems at ACA lengths, multi-threading isn't needed

The author has successfully solved virtually all ACA systems without multi-threading with great performance. There are a few systems that he hasn't solved; even for these, it is not obvious that multi-threading will help.

Multi-threading requires a little discipline to avoid horrible, hard to debug problems.

1. Any sort of constants or global lookup tables need to be "read only" or "never changed after the very beginning before any threads are kicked off". Otherwise, each thread must have their own copy in which to make their own changes. If initialized before the threads kick off, there must be something done before the kick off to make sure all the initialization is seen by all CPUs.
2. "Static" storage that changes needs to be avoided or at least locked. This is where many nasty bugs come from.
3. "Peeking" at shared static storage may require locking. For certain computer architectures, one must lock to both change and examine shared, changeable storage. Only if the storage never changes after initialization is it safe to "peek" at shared storage, provided proper locking has been followed when the shared object was created.

Object-oriented programming would make this a little easier. With objects, items that might reside in static storage end up as instance variables in the object, which gets rid of loads of problems. With that approach, a central thread creates all the storage needed for hill climbing (the CT, the PT, the partial keys, "pointers" to shared objects like tetragram counts). It makes N copies of this (objects make this easy) and then launches N threads to run their own, independent hill climb. The central thread waits until they are done and merges their ordered suite of answers into one big, final ordered suite. This can also be done with classic C type "structs" or whatever equivalent the language has, but objects were born for this. Note that these copies must be so-called "deep" copies – not just copying pointers to (what then turn out to be) common storage. No, every byte of actual data needs replication.

A protocol for initializing shared storage like a tetragram frequency list or a word list might look like this (pseudo code):

```
// sharedPointer is in static storage somewhere
if (sharedPointer is "null" or is "empty") {  // language dependent
    lockSomething();
    // Now that we're locked, make sure someone else isn't ahead of us
    if(sharedPointer is "null" or is "empty) {
       // only one thread ever gets to this point
       sharedPointer = InitializeSharedObject();
    }
    else;  // do nothing, some other thread already initialized it
    unlockSomething();  // make sure that even weakly consistent
                        // architectures propagate all changes.
}
return sharedPointer;
```

Note carefully that most of the time the first and last line of code are the only two executed. They will be in the body of some static code with a name like GetSharedPointerToTable or whatever kind of object is being dealt with. So, it costs almost nothing extra to get the pointer safely and yet no code ever sees anything but a fully initialized object.

One critical bit here is so-called “weakly consistent” architectures. The lock/unlock library code must ensure that all changes made while locked are propagated to all processors before the ‘unlockSomething()’ terminates. Most programming language lock primitives do this automatically, but check the rules and pick the right primitives.

Languages and threading

Language	**Threading and Locking Status**
C	Painful. Must create a “struct” with everything that is useful. Locking protocols usually come from the operating system or maybe POSIX; these do not move readily from one C environment to another. Threading primitives may also come from the operating system or POSIX.
C++	Objects make encapsulating easier. Later versions of C++ have threading capabilities. Use those.
Java	Built-in, easy to use.
Python	Basically, *does not do* threading.
Pascal / Delphi	Unclear – facilities appear to exist, but no firsthand experience. Has a form of objects.
Free Pascal	Facilities appear to exist, but OS dependent for best results. Has a form of objects.
C# / .NET / Visual Basic	Built-in, easy to use.
Javascript	“Web Workers” has at least partial support.

Topic Five: Processing ACA Digital Cons

The text in our digital cons on the cryptograpy.org website is very similar to what is published in the magazine. However, on close examination, it is not just US ASCII text. This leads to the potential for a variety of problems as code is easier when US ASCII can be assumed.

In a useful, modern touch, the Digital Cons files are encoded in UTF-8. So, just figure out how to tell your language to open the file with UTF-8. Otherwise, you may get garbage characters here and there.

Within the UTF-8 stream, alternative characters may appear.

1. Alternate quotation marks. Normally, in computer processing, one expects to see the ASCII single and double quote characters (') and ("). However, if the cipher was composed in a word processor, it is possible that some of the “other” quotation marks were used and end up in digital cons. Many of these have a “left” and “right” form. For example, these: “”.
2. The Norwegian/Danish “slashed O” (Ø) is sometimes used for number zero.
3. There is a “non blank space” which, while rare, needs to be translated to an ordinary space. Tab and Vertical Tab have not been seen, but would probably be best translated to a space.

4. Here's a snippet of Python which translates the known mischief makers to more expected ASCII values:
   line=line.replace("\u00D8","0").replace("\u0093",'"').replace("\u0094",'"')
   line=line.replace("\u201c",'"').replace("\u201d",'"')
   line=line.replace("\u0091","'").replace("\u0092","'")
   line=line.replace("\u2018","'").replace("\u2019","'").replace("\u00a0"," ")
5. Sometimes, especially in 6x6 ciphers, the number One (1) is underlined in the magazine. This convention disappears and does not appear in digital cons files.
6. Xenocrypts use the English alphabet, so no Spanish Tilde N or like things.

Once these are accounted for, it is easy to get Digital Cons read into one's code.

Topic Six: Performance

As it happens, the author was a senior computer performance specialist for nine years at a brand-name computer company. One learned surprising things about performance, its actual importance, and where to look for improvement.

The most important good news is that one can use any computer language or particular computer one is comfortable with. The fastest computer and the fastest language is not a requirement despite crypto code being hungry for performance. Use what you are good at – don't chase cycles and court frustration. It isn't needful.

The author has actually used Python3 on a fairly slow Android tablet. Together they almost certainly much slower (two, four, even ten times slower) than what the reader would be using. Yet, the combination was quite capable of solving all Aristocrats in an issue; one just waits a little longer for a batch process to finish. It can be done while one watches a movie or a sportscast, worst case.

Many ciphers solve by key exhaustion. A shrewd look at Cipher Exchange publication guidelines (followed in practice by most published crypts) have limits such as "10 to 15 times the length of the period." Moreover, actual published crypts, because of overall magazine space requirements, end up in the 100 to 150 character range. So, even 12 factorial is a long key in practice even if it is sometimes seen. Coding *and debugging* exhaustive schemes are much easier; prefer them if possible.

If one is still looking for "clever coding" then by far the most important thing to do is convert character strings to integer arrays. For a wide variety of reasons, there is a lot of practical efficiency gained. The conversion is a one-time cost if done ahead of the deep loops. Only the very best solutions need ever be translated back to character strings so one can see the answers.

Guidelines for Top Performance

1. Use FactorialKeySet for any exhaustive cipher that has a factorial-sized key space. The amount of time saved by doing FactorialKeySet (versus deep loops with a total cost of 10 to the 12^{th} and then throwing out the unqualified keys at period 12) will dwarf everything else we are discussing. It is far more important, in crypto, to figure out how to reduce iterations at the *design* level than with clever coding or compiled languages.
2. Theory matters. Favor algorithms with N log(N) or even log(N) performance and worry not so much about squeezing code out of subroutines. The former pays off much better.

3. Replace, in the deep loops, character strings with integer arrays (should be a one-time cost). A-Z have values 0 to 25, # is 26 (for Trifid et al.) or 26 to 35 are the digits 0-9 for 6x6 ciphers.
4. "Library" functions run typically no worse than N times log(N) where N is the total number of items in question. Some run at log(N). Using them even in interpretive code may well entirely outweigh the performance earned from a compiled language where one re-writes such functions ourselves. So, use generic lists or sort subroutines whenever you find them. The performance gains might be incredible and the coding and debugging saved also incredible.
5. Languages, such as Java or C#, using a "Just in Time" (JIT) compiler are often quite competitive. A comparison Patristocrat solver, coded as identically independently of language APIs as possible showed Java was only 20 to 40 per cent slower than C/C++. This is certainly close enough that its other benefits, if compelling to the coder, make it competitive.

The Ultimate Performance Calculation

The other thing to remember is that there is plenty of time. Suppose it averages ten minutes to automatically solve a cipher. Well, that is then only 1,000 minutes per issue as we have about 100 puzzles per issue. There are 1440 minutes per day. A ten year old computer running an interpretive language still stacks up well with this math. Use what you like; use what you understand and all will be well.

Appendix C. Useful Statistics for Solving

Cryptography, especially automated cryptography, is full of statistics. Surprisingly, only a handful of specific techniques will be required. The scoring, and its tetragram grading, have already been covered.

The next are universal in that they are applicable to by-hand or fully automated solving. They are known to apply to particular systems, but who knows? Maybe you will find a new use for some of these or even novel ones. You don't have to be an expert to apply these. Just understand them well enough to program and then to apply to cryptograms.

Index of Coincidence (abbreviated IC or IoC)

This is easy to describe and implement. Many use it for periodic ciphers, but the author prefers the more specialized Kasiski method for periodics. The text is presumed to be alphabetic.

The IoC can also be used to help identify cipher systems as there are published IC values for languages (if the IC matches what is predicted for plaintext in that language, it points to simple substitution or transposition, for instance).

Pseudo code:

```
double IndexOfCoincidence(int freqArray[]) {
    double sum = 0.0;
    double tot = 0.0;
     for (int i=0; i< freqArray.length; ++i ) {
         sum = sum + freqArray[i]*(freqArray[i]-1)
         tot = tot + freqArray[i];
     }
    double denominator =  tot*(tot-1.0);

    if ( denominator > 2.0) // non-degenerate cases
            return sum/denominator;
    else return 0.0;   // 0.0 for degenerate cases (very small data)
}
```

Really, that's all there is to it.

Here's some published Index of Coincidence data from our Xenocrypt Handbook:

Source	Value
Random	0.0385
English	0.0661
Italian	0.0738

Source	Value
German	0.0762
Spanish	0.0775
French	0.0778
Portuguese	0.0791

Kasiski Method

This directly tests for coincidences in ciphertext. The concept is that if you see a ciphertext pair like this: YM and then YM again appears either six or twelve or any multiple of six apart, it is presumed a "hit" for the cipher actually being period six. Coincidences are possible, so this isn't infallible, but in practice, counting the "hits" gives a fairly effective idea of what the period is; effective enough to inform an auto solver. One problem is that if the period is twelve, then periods like 3, 4, and 6 may also score as well or nearly as well.

More weight could be given to "triples" where YMX in the ciphertext appears at the specified period's interval, but the most basic coding style will give "double credit" for both YM and MX, so even a naïve looking implementation gives a premium for triples, quadruplets, etc.

Pseudo code:

```
int KasiskiMethod(int period, string ct) {
    int coincidence = 0;
    lenM1 = ct.length – 1;  // stop short of very end because we
process letter pairs
    for (int i=0; i<lenM1; ++i) {
         char base1 = ct[i];
         char base2 = ct[i+1];
         // Note carefully how "j" is handled.
         for (j=i+period; j<lenM1; j+=period) {
            char comp1 = ct[j];
            char comp2 = ct[j+1];
            if (base1==comp1 and base2 == comp2) {
                coincidence=coincidence+1;
            }
         }
    }
    return coincidence;
}
```

This is probably called once per "interesting" period (perhaps from 3 to 15 or so for ACA systems). The individual results can be stored in an ordered list and used in that order or perhaps with an eye to favoring larger periods over smaller ones at the same magnitude.

An override parameter to periodic systems to specify a period is often a good idea because it may get tedious (if no period is specified) to see the code process two to four periods of nearly equal magnitude. It is also the case that periods are sometimes given in a tip.

The Chi Square Statistic

This is a very general statistic and may have other uses we have not identified. It's known use in ACA cipher systems is for Bifid and Trifid to predict their periods. Bifids have an alphabet of 25 letters and Trifids have 27 (the # character is in the ciphertext). Other than that, the ideas are pretty much the same.

There are several write-ups and even, it appears, alternative calculation methods. This version works very well for ACA ciphers and is easy to understand. It uses a version best known as the Chi Square test for independence.

Here's a sketch in Python. An important detail, how to "bucketize" the ciphertext into "buckets" based on rows and columns (Bifid) and based on depth, rows, and column (Trifid) is assumed to have been performed elsewhere. See SHMOO and O'PSHAW articles (JA95, SO95) in the archive to deal with this effort and also to obtain data for a unit test.

```
import io
# Chi-square
#
# Chi-square is calculated as follows for Bifid or Trifid (see also SHMOO and
#  O'PSHAW articles in Cm):
#  1. Bucketize the input and count the frequency of each letter
#     (including # in Trifid) and put that in an array.
#     These are the "observations".  The bucketizing comes from
#     laying out the cipher with the indicies as cipher letters (rows
#     and column indicies for Bifid, depth, row, and column
#     for Trifid).  It must be done for each period.
#     In Trifid period 8, the five buckets are as follows from each
#     unit of the ciphertext that is a full
#     multiple of the period.  The understanding, per period,
#     can be precomputed, making implementation (not shown) easy:
#       a. the first two cipher letters in each block contain only
#          depth indicies.
#       b. The third cipher letter (in period 8) has either one or two
#          plaintext depth indicies and two or one row indicies.
#       c. Letters four and five, in period 8, contain plaintext row
#          indicies only.
#       d. One letter (letter six in period 8) will transition
#          between row and column indicies with
#          some from one and some from the other.
#       e. The remaining letters (letters seven and eight in period 8)
#          have only column indicies.
#  2. Calculate the Expected value of each observation.  This is done
#       by doing a horizontal
#       and vertical sum of the buckets (R and C).
#       Then, Expected[i,j] = (R[i] * C[j])/cTextLength where R[i] is
#       the "i-th" bucket an C[j] is the "j-th" letter frequency column.
#  3.  The Chi Square is then the sum of each
#      Square(Observation[i][j]-Expected[i][j])/Expected[i][j]
#      . . .where Square is the number multiplied by itself.
#
#      Note there can be terms where Expected[i][j] is zero, usually
#      because the column sum, C[j], is all zeroes (for instance,
#      cipher Z never appears in the ciphertext of O'PSHAW's example).
#      If that happens, simply skip the term.
#
# This ChiSquare test does not care if it is a Trifid or Bifid test.
```

```
#   It only cares that someone computed the "bucket"
#   frequency count correctly for the current period.
def ChiSquare(bucket) :
    # First, calculate the Row matrix, R, the sum of each row item.
    R = []
    grandTot=0   # ciphertext length, derived from sum of
                 # all bucket elements.
    for i in range(0,len(bucket)) :
        totV=0
        for j in range(0,len(bucket[0])) :
            totV=totV+bucket[i][j]
        R.append(totV)
        grandTot=grandTot+totV
    # Now calculate C, the Column matrix
    C=[]
    for j in range(0,len(bucket[0])) :
        totH=0
        for i in range(0,len(bucket)) :
            totH=totH+bucket[i][j]
        C.append(totH)
    #  The passed "bucket" is the Observation matrix.
    #  With R and C, calculate the Expected matrix, E
    E = []
    for i in range(0,len(bucket)) :
        E.append([])
        for j in range(0,len(bucket[0])) :
            # E[i][j] = R[i]*C[j]/gradeTot in other languages.
            E[i].append( R[i]*C[j]/grandTot )
    # Now calculate the Chi Square itself
    chiSquare = 0.0
    for i in range(0,len(bucket)) :
        for j in range(0,len(bucket[0])) :
            if(E[i][j]!=0) : # skip any terms where E[i][j] is zero
                diff = bucket[i][j]-E[i][j]
                chiSquare=chiSquare+(diff*diff)/E[i][j]
    return chiSquare
```

More extensive code, which includes a unit test based on O'PSHAW's examples, was omitted in the interest of space. But, with care, O'PSHAW's data can be captured and used to test your implementation. Just leave off the summary data.