MIKICalculator and Password generator

1. Main MikiCalculator and Password generator idea and solution

I don’t know if anyone made something like my solution, if that concept is used anywhere. This is entirely my concept of the password generator that I decided to implement. It wasn't easy, but it works.

Each password is a decimal number, but in n-numbering system e.g. if we think about hexadecimal, and password is “ABCD1234” then decimal number for this password is “2 882 343 476”. The problem is, when password is much longer, and password consists of much more characters, e.g. 62-numbering system, 32 characters in password. In this situation last password is “zzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzz”, in decimal it is “2 272 657 884 496 751 345 355 241 563 627 544 170 162 852 933 518 655 225 855”. That is much more above decimal variable, ignoring the fact that calculation on decimals are very slow. So, what we can do… How to make a calculation on that huge numbers??? Simply 😉

First of all, Let's assume that the number is a string, so we can build a number with 2,147,483,647 digits. I think, that we can make it longer, by dividing longer strings to shorter, but now I don’t need it. Now, if we have two “string-numbers” we can make a scholar calculation, digit by digit, that’s easy, but it can take a lot of time. What can we do to make it faster.

Biggest and fastest variable we can use is 62bit-integer (long), let’s use it. Divide each string-number to array or list of longs. I prefer List, lots of operation on lists are faster.

We must do it from back side, so we have:

List<long> Dig1 = new () {2272, 657884496751345355, 241563627544170162, 852933518655225855}

If we have two long-list-digits like this above, we can make all scholar calculations on them, but now we don’t take each digit separately, we take whole part-number, rules of scholar calculation are still the same. Two thigs are worth noticing. Long variable can be max 9,223,372,036,854,775,807 – 19 digits, but I use List with 18 digits for two reasons. First reason is, if we make addition on longer number then 18 digits, we can exceed max value of long, second – multiplication, we can’t do it on 18-digits longs for the reason above, but we can on 9-digits, in worst scenario it can give us 18-digits and never exceed long max value. That’s why multiplication works on 9-digits long, and after calculation it can be or not converted to 18-digits list. Final result of these calculation is returned in List<long>.

Before describing dividing, which is much complicated than addition, subtraction and multiplication, is worth to mention, that if we have lists, we can do all calculation on them without converting to string-number, and if we can build list without string, we can do it all the time. You can convert list to string if you have the final result. That is much faster.

Dividing... I used something between scholar calculation and repeated subtraction. Repeated subtraction can be very slow, especially if we want to divide big number by small number. That’s why I used something that I called “SafeMultiplier”, it is a number witch is result of divisor multiplication, and has value very close to divident. This “SafeMultiplier” we can subtract from dividend, this operation is a substitution of multiple subtraction. Now, if SafeMultiplier is result, multiplicand is divisor, so multiplier (not SafeMultiplier) is a part of our final quotient, we must add all partial-quotients to have final quotient.

How to calculate SafeMultiplier…

Imagine, we have two numbers, 9876545 and 32345, we want to divide: 9876545/32345 == 305 and something 😉.

We work on the lists, we can’t take whole list, we must take part of both lists. Simplifying, for numbers above I will take first 4 digits from dividend and 3 from divisor. Why??? Always divisor is bigger then divider, if not there is nothing to do, we have quotient 0 and the rest (modulo) == dividend.

So, we take 9876 and 323. Because we take part of numbers, we don’t know what digits are next, we must set up worst scenario, for dividend is 0, for divisor is 9, but still we can’t take next digit, so it’s better to subtract 1 from dividend and add 1 to divisor. Of course, if we take whole number, we do not have to do it, we know whole number. So, we don’t know, we add and subtract… then divide… we have part of SafeMultiplier

9875 / 324 = 30 , look above, 305… now…

Simplifying, if we put original numbers together, and align them to the left, we can see, where we should put our part of SafeMultiplier (green), but still one place left (red), this place will be 0:

000030? ---------🡪 300 (multiply by 10, or/and add zeros to list)

\_\_\_\_\_\_\_\_

9876545

32345

Next step, multiply 30 and original 32345 == 970350. Due to the fact that want to add 0 (multiply by 10) to 30, we must add 0 to 970350 too, so we have 300 and 9703500. Now subtract 9703500 from original 9876545 == 173045.

173045 > 32345, so next loop….

Repeat steps above, but with 173045: 1729 / 324 == 5 and something… it is next part-quotient. Add part-quotients == 305, that’s it. Let’s calculate rest (modulo)…

5\*32345 == 161725

173045 – 161725 == 11320 ---- > 11320 < 32345 ---🡪 end of calculation, and 11320 is our rest (modulo). Final quotient is 305 and 11320/32345.

Final result of division is returned in List<List<long>> where: List[0] is quotient, List[1] is modulo.

Easy, isn’t it???

1. Lehmer algorithm and password-number.

As I wrote above, each password is a number, we can write it as password-decimal-number (password-number).

To generate all passwords we can iterate from 0 to n, but with huge amount of combinations it seems to be not quite good. Passwords e.g. aaaaaaaaaaa0, aaaaaaaaaaa1 can be good, but rarely, I think. So let’s play lottery. Question is, how generate random passwords and avoid repetitions? Lehmer algorithm is the answer. When talking about numbers…

Xn == (a \* Xm + I) mod P --🡪 here there is no sense to use “a” (offset) so remove “a”

Xn == (Xm + I) mod P

Xn – Next numer

Xm – Previous number (seed)

I – Increment

P – Pool (passwords, numbers)

Let’s take the pool of password-numbers 0-10, 0 is password too. We have pool of 11 passwords P==11. If we set Increment == 1, order of generated password-numbers will be:

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 and again 0, 1 ….. but if we set Increment == 3

0, 3, 6, 9, 1, 4, 7, 10, 2, 5, 8 and again 0, 3 …... let’s call it Lehmer circle.

As you see, we close the Lehmer circle when all numbers are done or we return to the first in circle.

Two important rules for Increment. Increment must be a prime number and P % I != 0. If these conditions are met, all generated numbers will be unique until we exhaust the pool. But what with huge numbers??? We can use power of Increment, but still, base must be a prime number, and final Increment must meet the condition P % I != 0.

Sometimes, especially when masks are used we want to avoid part of the pool, how to do it… There is another way to count Xn, we don’t need Xm. If we count generated numbers (Iterations) from 0 to n, than:

Xn == (Iteration\*Increment) mod P

Iteration == Password number, but 0 is number too, so password0 is 0.

If we want password 3, Increment is 1, P==11, password3 == 3\*1 mod 11 == 3

If we want password 3, Increment is 3, P==11, password3 == 3\*3 mod 11 == 9 e.t.c.

In this manner, if we don’t change P and I, we can check and save which passwords are done and restore sessions. We can divide password pool on smaller ones, and run these pools on different machines.

1. Conversions. Password-number -> password, password -> password-number.

It seems nothing special, number to password can be done by repeated division, like DecToBin, password to number by multiplication and addition, both can take a lot of time and sources.

Faster method, which brings additional benefits if we want to use masks, is table method. Of course, I use List.

List<List<List<long>>> ConvTable where:

ConvTable[CharPos][CharValue] = DigList.

And

List<List<char>> MaskTable where e.g.:

MaskTable[CharPos] = {0, 2, A, z}. It means, that 0=0, 2=1, A=3, z=4. Chars are only signs to describe any number, and they are sorted in a growing way. On different CharPos the same char can have different value.

CharPos is a character position in password (from right)

CharValue is position in MaskTable[CharPos] where we can check, what number is represented by this character. If we know this two things we can take DigList from ConvTable.

When converting from password to password-number we must find in ConvTable proper CharValue on proper CharPos for all characters, take DigLists, and add them all together.

When converting from password-number to password, we must find highest DigList which is lower or equal to password-number, subtract it from password-number, find next which is lower or equal to what left e.t.c. to 0. Here we must search from MaxCharPos.

It seems to be easy, but there are situation it’s not. We used to calculate on mononomial numbering system. If we use masks, mononomial system can produce password-numbers which don’t meet mask. Passwords should be discarded, it is waste of time. What to do???

Use polynomial numbering system. You can ask, are polynomial systems exists??? Yes, e.g. Dozen, Small gross, Gross… If we have ConvTable we can convert and use different monomial system on different position, even unary system. Unary system is a little tricky. As a rule, there is no zero in it. It has sign or not, how to write no sign, but it is not only one problem. Problem appears when we try convert password-number to password, let’s try, but first I tell you, the first problem is easy to solve. If we think about first password, which is 0, or something else, I mean that in mask on first position from right is only one character, then if there is no sign, there is no password. This way we must assume, that in unary zero exists, but only on this, first position…

Mask:

?1=A (0)

?2=BCz (0, 1, 2)

?3=F (1)

?4=xz (0, 1)

With this mask we can make passwords and password-numbers:

A and BA = 0

CA = 1

zA = 2

FBA and xFBA = 3

FCA and xFCA = 4

FzA and xFzA= 5

zFBA = 6

zFCA = 7

zFzA = 8

We now can tell, that we have our pool (P) == 9, A = 0 is password too.

One thing before next steps… In constant length passwords if we have password-number, after conversion to password we must fill it to full length with zeros from right to left, (abcd -> 0000abcd) then add to passwords list, because only zeros will not occur in higher passwords. The same if non-constant passwords we want, but we add password step, by step. We add one zero, and add to passwords list, next zero, and add to passwords list. E.t.c.

Think about unaries… if we are adding zeros to first password (A = 0) we fist get BA, it’s ok, if we want to add next zero we will meet mask ?3, where there is no zero, we will get FBA, but F is not zero, so F makes different password-number, FBA=3, of course, we can do this, but 3 will appear in next iterations, so it does not make sense, repeated password, we should break adding, and generate next password-number.

On the other hand, if password is constant length, and this is the point of the matter, first passwords can’t exists without ?3 mask. In this example there are only 3 password-numbers to refuse (0, 1, 2), but imagine that unary is on ?32 or there is more mask like this??? There will be a lot passwords to refuse. Let’s look again on this example and write down first possible constant length password, it is xFBA (0, 3, 0, 0) = FBA = 3. We can start from 3, fill with zero (x), add to list, then 4 = FCA, fill with zero (x), add to list ……. And end with zFzA = 8. This way, all passwords will fit to the mask.

Technically, when generator starts, it builds MaskTable, checks Increment, then collects from MaskTable[CharPos] first [0] characters, from each CharPos, and last [^1] from each CharPos. From fist characters on each CharPos it builds first possible password, from last – last possible password, then convert both to password-number. First password is the point where we start, the last – where we end, but if we add to it 1, because, 0 is password to, so Pool (P) is last possible password + 1. As I wrote before, we can count all password or check if we return to 0 to break when all is done. How to start from point of start??? Use this Xn == (Iteration\*Increment) mod P

That’s why we must count passwords, we can say that the Iteration is the password number.

Let's return to the unary system problems.

Unary system makes one more problem when converting from password-number to password. That’s why I make List<List<long>> AllUnarysValueList which is similar to ConvTable, but on each CharPos it has a DigList which value is lowest password-number value starting from this CharPos to place of ones. (unaries/unarys…??? I don’t know how to write it 😉 ). Remember, if I write about password I describe position in order from place of thousands to places of ones:

?6?5?4?3?2?1, but values, DigLists, characters in Lists from first position (?1) are always on [0], that’s why Lists looks inverted, in other words, places of ones are on [0].

ConvTable from this example looks like this:

A B C z F x z

{ {{0}} {{0}{1}{2}} {{3}} {{0}{3}} } numbers are DigLists, but in this example numbers are to small.

AllUnarysValueList is:

{ {0} {0} {3} {3} } (Unary:A=0, The same value = 0, Unary:FBA=3 New value, The same value = 3)

Let’s try to convert password-number = 2. We look for DigList higher then 2 from “xz” position to “A” position, but still we look for DigList lower than or equal 2. We find 3, 3 is bigger, then we take lower Diglist – 0, in this loop we don’t take zeros, we want to find FirstNonZero, we will add zeros later, so there is no char. Next loop iteration will find 3 (F), we still look for lower, so no char. Next we will find 2 (z) it’s equal, so we take it and subtract 2-2==0, next step we look for 0, there is only one value, 0 == A, so 2 = zA.

If you try to do it with 4, it will not work. It should be FCA. Let’s try, from CharPosMax=3 (password length -1)…

4 is our ToCompare, we will find z(3), 4-3=1, CharPossMax--, WRONG… what to do…

AllUnarysValueList is now necessary. Looking for 4 if in password unarys exists, we must first check, if 4 is bigger than value on CharPos in AllUnarysValueList. We look at position 4 AllUnarysValueList [3], there is 3, 4 is bigger so we must subtract 4-3=1. This is our ToCompare. We left the info, that unary is subtracted and info, that from this position converter must add char of all lower positions unaries. Now we compare 1 with whole ConvTable. We find 3 (z) we take lower value, x, but x is 0, so don’t take it and do next step, CharPosMax--. Still we have 1, but we met unary F=3, we subtracted it before and converter know, that must take it, so first char=F, still we have 1. CharPosMax--. We find 1 (C), subtract 1-1 = 0, take it, CharPosMax--, we find 0 (A), finnaly we have FCA. Now all is good.

Almost. Lists compare is relatively long procedure. I made it faster.

Before we start to compare password-number (List) with ConvTable Lists, we can simply compare two longs, that contains the information about those two Lists. This way we can make searching loop shorter if we know where to start. I think, that we can make shorter in this way higher loop to, but now I get errors if I turn on this option. So…

Imagine long number which can tell us two things… How many digits contains string-number-password, and something about this string-number-password value e.g. first four digits from left. Let’s take 3456543……………….. (60 digits). We can build long value in format XXXX…..YYYY, where XXXX…… is the full length of our string-number-password and YYYY are first 4 digits, so… 60\*10000 + 3456 = 603456. We can fast and easily build that number from Lists and compare them to make internal loop shorter. I have checked it. Without it, when on position are 62 characters, converter must in worst scenario make 62 iterations with 62 list comparisons (it takes a lot of time), but if we compare these longs there are only 1 or 2 iterations and comparisons.

Why long??? I assumed that no one would check passwords longer then max string length, so max string length we can describe with 10 digits, and I think, that first four digits from string-password-number are sufficient in terms of information about the whole number. Long may have safely value with 18 digits, in my solution, highest long number can be:

2147483647\*10000 + 9999 = 21474836479999, it is still safe, below long max value.

And that’s all about main concept and algorithms.

I have written this program to decode QLocker, I know, little chance, but I have a lot of fun. Moreover, I hope that my work will be useful to someone.

I think, all in my program is full configurable, but mainly focused on decoding 7z ( with HC, JTR, 7z). You can change settings in Config.txt and use HC for any type of hash you want, but if you don’t change the code, as HC/JTR finds password, program will try to check password with 7z, so put in the Coded folder any 7z file. This way, all found passwords will be saved as FalsePositive……txt, but for you these password can by TruePositive.

If you decide to make some modifications in MikiCalculator, I wrote Random Calculator. Random Calculator generates random string-numbers and makes two way calculations. If something is going wrong, the program will inform about the error.

I you decide to make some modifications in Passwords Generator, always check it with Test Generator.

Of course, there is simply string-number calculator too. Use it as you wish.

Good luck.