```
        podstaw char za kolejną literkę w haśle
        wywnioskuj na podstawie znanych ograniczeń jak najwięcej innych znaków (funkcja filling_pass). Jeśli wyjdzie
```

Odpaliliśmy więc nasz algorytm:

```
C:\Users\xxx\Code\RE\CTF\2015-10-02 def\crypto300>python hackz.py
Cpgl5bpyy5qz{p5lz`5VAS5eb{pg;5[zb5\5}tcp5az5r|cp51z`5a}p5gpbtgq5szg5tyy5a}|f5}tgq5bzg~5zg5xtlwp5r`pff|{r;5A}p5sytr5|f
Izmf?hzss?{pqz?fpj?\KY?ohqzm1?Qph?V?w~iz?kp?xviz?fpj?kwz?mzh~m{?ypm?~ss?kwvl?w~m{?hpmt?pm?r~f}z?xjzllvqx1?Kwz?ys~x?vl
Qbu~'pbkk'chib'~hr'DSA'wpibu)'Ihp'N'ofqb'sh'`nqb'~hr'sob'ubpfuc'ahu'fkk'sont'ofuc'phul'hu'jf~eb'`rbttni`)'Sob'akf`'nt
Rav}$sahh$`kja$}kq$GPB$tsjav*$Jks$M$lera$pk$cmra$}kq$pla$vasev`$bkv$ehh$plmw$lev`$skvo$kv$ie}fa$cqawwmjc*$Pla$bhec$mw
S`w|%r`ii%ajk`%|jp%FQC%urk`w+%Kjr%L%mds`%qj%bls`%|jp%qm`%w`rdwa%cjw%dii%qmlv%mdwa%rjwn%jw%hd|g`%bp`vvlkb+%Qm`%cidb%lv
Tgp{"ugnn"fmlg"{mw"AVD"rulgp,"Lmu"K"jctg"vm"ektg"{mw"vjg"pgucpf"dmp"cnn"vjkq"jcpf"umpi"mp"oc{`g"ewgqqkle,"Vjg"dnce"kc
Ufqz#tfoo#glmf#zlv#@WE#stmfq-#Mlt#J#kbuf#wl#djuf#zlv#wkf#qftbqg#elq#boo#wkjp#kbqg#tlqh#lq#nbzaf#dvfppjmd-#Wkf#eobd#jp
Very well done you CTF pwner. Now I have to give you the reward for all this hard work or maybe guessing. The flag is
Wdsx!vdmm!enod!xnt!BUG!qvods/!Onv!H!i`wd!un!fhwd!xnt!uid!sdv`se!gns!`mm!uihr!i`se!vnsj!ns!l`xcd!ftdrrhof/!Uid!gm`f!hr
Xk|w.ykbb.ja`k.wa{.MZH.~y`k| .@ay.G.foxk.za.igxk.wa{.zfk.|kyo|j.ha|.obb.zfg}.fo|j.ya|e.a|.cowlk.i{k}}g`i .Zfk.hboi.g}

C:\Users\xxx\Code\RE\CTF\2015-10-02 def\crypto300>
```

Świetnie - zdobyliśmy flagę - `cryptanalysis_is_hard`

ENG version

Very interesting task for us, we had to put much effort into this by we finally solved it.

We get a ciphertext:

```
320b1c5900180a034c74441819004557415b0e0d1a3169180118455241473 84f5700264f48091e45
00110e41030d1203460b1d0752150411541b455741520544111d0000131e0159110f0c16451b0f1c
4a74120a170d460e13001e120a1106431e0c1c0a0a1017135a4e381b16530f330006411953664334
593654114e114c09532f271c490630110e0b0b
```

And code whcih was used to encode it (rewritten to Python since we're not fans of PHP):

```
def encrypt(plainText):
    space = 10
    cipherText = ""
    for i in range(len(plainText)):
        if i + space < len(plainText) - 1:
            cipherText += chr(ord(plainText[i]) ^ ord(plainText[i + space]))
        else:
            cipherText += chr(ord(plainText[i]) ^ ord(plainText[space]))
        if ord(plainText[i]) % 2 == 0:
            space += 1
        else:
            space -= 1
    return cipherText
```

The biggest issue here is the moving space. However its state depends only on the lowest bit of the plaintext character, which we exploited to solve the task. The solution was split into two parts - first we extracted lowest bits of the plaintext and after that we decoded the whole ciphertext.

First attempts to extract the lowest bits:

```
def verify(cipherText, guessedBits):
    space = 10
    for i in range(len(guessedBits)):
        if i + space < len(cipherText) - 1:
            if i + space >= len(guessedBits):
                return True
            if (ord(cipherText[i]) & 1) != ((ord(guessedBits[i]) & 1) ^ (ord(guessedBits[i + space]) & 1)):
                return False
        else:
            if space >= len(guessedBits):
                return True
            if (ord(cipherText[i]) & 1) != ((ord(guessedBits[i]) & 1) ^ (ord(guessedBits[space]) & 1)):
                return False
        if guessedBits[i] == '0':
```

```
            space += 1
        else:
            space -= 1
    return True


def decrypt(cipherText, guessedBits, i):
    if i >= len(cipherText):
        print 'ok:', guessedBits
        return
    if len(guessedBits) == 10:
        print (int(guessedBits, 2) / 1024.0) * 100, '%'
    if verify(cipherText, guessedBits):
        decrypt(cipherText, guessedBits + '0', i + 1)
        decrypt(cipherText, guessedBits + '1', i + 1)
```

This is a simple brute-force with prunning - we test all possibilities and if verify() failed we prune given branch.

Unfortunately this was too slow. We started with some optimization of the code - and if I remember correctly this resulted in 600 times faster execution (!) (and we were already running on pypy for all the tests):

```
def verify(cipherText, guessedBits, length, guessed_len):
    space = 10
    for i in range(guessed_len):
        if i + space < length - 1:
            if i + space >= guessed_len:
                return True
            if (cipherText[i] & 1) != ((guessedBits[i] & 1) ^ (guessedBits[i + space] & 1)):
                return False
        else:
            if space >= guessed_len:
                return True
            if (cipherText[i] & 1) != ((guessedBits[i] & 1) ^ (guessedBits[space] & 1)):
                return False
        if guessedBits[i] == 0:
            space += 1
        else:
            space -= 1
    return True


def decrypt(cipherText):
    guessed_bits = [0] * len(cipherText)
    length = len(cipherText)
    i = 0
    orded_cipher = [ord(c) for c in cipherText]
    decrypt_r(orded_cipher, guessed_bits, i, length)


def decrypt_r(orded_cipher, guessedBits, i, length):
    if i >= length:
        print 'ok:', guessedBits
        return
    if i == 10:
        print (int(''.join(str(c) for c in guessedBits[:10]), 2) / 1024.0) * 100, '%'
    if verify(orded_cipher, guessedBits, length, i):
        guessedBits[i] = 0
        decrypt_r(orded_cipher, guessedBits, i + 1, length)
        guessedBits[i] = 1
        decrypt_r(orded_cipher, guessedBits, i + 1, length)
```

This was already reasonably fast. We fired this four times for different prefixes on one machine.

But since we had to wait anyway, we decided to try a different approach (viva la algorithmics). Instead of verifying the whole password every time (by going forward) we rejected impossible solutions right away:

```
def decrypt(cipherText):
    guessed_bits = ['?'] * len(cipherText)
    length = len(cipherText)
    i = 0
    orded_cipher = [ord(c) & 1 for c in cipherText]
    decrypt_r(orded_cipher, guessed_bits, i, length, 10)
```

```
def try_guess(orded_cipher, guessedbits, i, length, guess, space):
    guessedbits = list(guessedbits)
    guessedbits[i] = guess
    if i + space < length - 1:
        nextndx = i + space
    else:
        nextndx = space

    nextbit = orded_cipher[i] ^ guess
    if guess == 0:
        newspace = space + 1
    else:
        newspace = space - 1

    if guessedbits[nextndx] == '?' or guessedbits[nextndx] == nextbit:
        guessedbits[nextndx] = nextbit
        decrypt_r(orded_cipher, guessedbits, i + 1, length, newspace)

def decrypt_r(orded_cipher, guessedbits, i, length, space):
    if i >= length:
        print 'ok:', ''.join(str(c) for c in guessedbits)
        return
    if guessedbits[i] == '?':
        try_guess(orded_cipher, guessedbits, i, length, 0, space)
        try_guess(orded_cipher, guessedbits, i, length, 1, space)
    elif guessedbits[i] == 0:
        try_guess(orded_cipher, guessedbits, i, length, 0, space)
    elif guessedbits[i] == 1:
        try_guess(orded_cipher, guessedbits, i, length, 1, space)
```

We use a tri-value-boolean in guessedbits, 0 for `there is definitely 0`, 1 for `there is definitely 1` and ? for `don't know`.

It took us an hour to write this and debug but we got the results within a few seconds (while the previous version was still computing).

Anyway, the final correct sulutions (for lowest bits of the plaintext) were four:

sln =
'10010110011101011100101000101101100100000100001001110100101110100101001111000010011100000111100100111101001010010110011010100011011010100011011010' sln =
'10010110011101011100101000101101100000100010111010011010000011001001100111100111011110000000011010000111101001010010110011010100011011010100' sln =
'01010110000101011101000010100001101001010010111010111000100111000010010000011001000110101001110101111110100001000110110101001010111111110100' sln =
'01010110000101011101000010100001101001010010110000110001001110001100100011000110011101010000101000011101100111000100101001010111001001011'

This way we got a set of equations with 140 variables. We tried to use a constraint-programming solver to solve it. For example for the bits number 4:

```
from constraint import *
problem = Problem()

ODD = range(33, 128, 2) + [13]
EVEN = range(32, 128, 2) + [10]

problem.addVariable(0, ODD)
problem.addVariable(1, EVEN)
problem.addVariable(2, EVEN)
problem.addVariable(3, ODD)
problem.addVariable(4, EVEN)
# (...) snip
problem.addVariable(134, ODD)
problem.addVariable(135, EVEN)
problem.addVariable(136, ODD)
problem.addVariable(137, EVEN)
problem.addVariable(138, EVEN)

problem.addConstraint(lambda av, bv: av ^ bv == 0x32, (0, 10))
problem.addConstraint(lambda av, bv: av ^ bv == 0xb, (1, 10))
problem.addConstraint(lambda av, bv: av ^ bv == 0x1c, (2, 12))
problem.addConstraint(lambda av, bv: av ^ bv == 0x59, (3, 14))
problem.addConstraint(lambda av, bv: av ^ bv == 0x0, (4, 14))
```

```
problem.addConstraint(lambda av, bv: av ^ bv == 0x18, (5, 16))
# (...) snip
problem.addConstraint(lambda av, bv: av ^ bv == 0x6, (133, 17))
problem.addConstraint(lambda av, bv: av ^ bv == 0x30, (134, 16))
problem.addConstraint(lambda av, bv: av ^ bv == 0x11, (135, 15))
problem.addConstraint(lambda av, bv: av ^ bv == 0xe, (136, 16))
problem.addConstraint(lambda av, bv: av ^ bv == 0xb, (137, 15))
problem.addConstraint(lambda av, bv: av ^ bv == 0xb, (138, 16))

print problem.getSolutions()
```

Unfortunately, it was taking a long time (and we're not sure if it would compute in this century). So in the meanwhile we attempted to make a custom solver:

```
import string
slv = [None] * len(sln)

def filling_pass(slv):
    while True:
        any = False
        space = 10
        for i in range(len(sln)):
            if i + space < len(sln) - 1:
                nx = i + space
            else:
                nx = space
            if sln[i] == '0':
                space += 1
            else:
                space -= 1
            if slv[i] is not None:
                sn = ord(slv[i]) ^ ord(ciph[i])
                if slv[nx] is None:
                    slv[nx] = chr(sn)
                    if (sn >= 32 and sn < 127) or sn == 10 or sn == 13:
                        any = True
                    else:
                        return False
                else:
                    if slv[nx] != chr(sn):
                        return False
        if not any:
            return True

def tryit(slvo, start):
    while slvo[start] is not None:
        start += 1
        if start >= len(slvo):
            print ''.join(' ' if c is None else '.' if ord(c) < 32 else c for c in slvo)
            return
        continue
    for c in string.printable:
        slv = list(slvo)
        slv[start] = c
        possible = filling_pass(slv)
        if possible:
            tryit(slv, start)

tryit(slv, 0)
```

And again for the second time in this task, what was taking hours in the naive implementation, took only *seconds* with a better algorithm. It's nice to think that even though we get more and more powerful computers, thinking about the computational complexity pays off sometimes.

The solver worked like this:

```
plaintext = ['?'] * plaintext_length
while (not all characters in plaintext are decoded):
    for char in charset:
        use char as next letter in password
                using known constraints try to figure out as much as possible of next characters in plaintext (fillir
```

This way we got:

```
C:\Users\xxx\Code\RE\CTF\2015-10-02 def\crypto300>python hackz.py
Cpgl5bpyy5qz{p5lz`5VAS5eb{pg;5[zb5\5}tcp5az5r|cp5lz`5a}p5gpbtgq5szg5tyy5a}|f5}tgq5bzg~5zg5xtlwp5r`pff|{r;5A}p5sytr5|f
Izmf?hzss?{pqz?fpj?\KY?ohqzm1?Qph?V?w~iz?kp?xviz?fpj?kwz?mzh~m{?ypm?~ss?kwvl?w~m{?hpmt?pm?r~f}z?xjzllvqx1?Kwz?ys~x?vl
Qbu~'pbkk'chib'~hr'DSA'wpibu)'Ihp'N'ofqb'sh'`nqb'~hr'sob'ubpfuc'ahu'fkk'sont'ofuc'phul'hu'jf~eb'`rbttni`)'Sob'akf`'nt
Rav}$sahh$`kja$}kq$GPB$tsjav*$Jks$M$lera$pk$cmra$}kq$pla$vasev`$bkv$ehh$plmw$lev`$skvo$kv$ie}fa$cqawwmjc*$Pla$bhec$mw
S`w|%r`ii%ajk`%|jp%FQC%urk`w+%Kjr%L%mds`%qj%bls`%|jp%qm`%w`rdwa%cjw%dii%qmlv%mdwa%rjwn%jw%hd|g`%bp`vvlkb+%Qm`%cidb%lv
Tgp{"ugnn"fmlg"{mw"AVD"rulgp,"Lmu"K"jctg"vm"ektg"{mw"vjg"pgucpf"dmp"cnn"vjkq"jcpf"umpi"mp"oc{`g"ewgqqkle,"Vjg"dnce"kc
Ufqz#tfoo#glmf#zlv#@WE#stmfq-#Mlt#J#kbuf#wl#djuf#zlv#wkf#qftbqg#elq#boo#wkjp#kbqg#tlqh#lq#nbzaf#dvfppjmd-#Wkf#eobd#jp
Very well done you CTF pwner. Now I have to give you the reward for all this hard work or maybe guessing. The flag is
Wdsx!vdmm!enod!xnt!BUG!qvods/!Onv!H!i`wd!un!fhwd!xnt!uid!sdv`se!gns!`mm!uihr!i`se!vnsj!ns!l`xcd!ftdrrhof/!Uid!gm`f!hr
Xk|w.ykbb.ja`k.wa{.MZH.~y`k| .@ay.G.foxk.za.igxk.wa{.zfk.|kyo|j.ha|.obb.zfg}.fo|j.ya|e.a|.cowlk.i{k}}g`i .Zfk.hboi.g}

C:\Users\xxx\Code\RE\CTF\2015-10-02 def\crypto300>
```

Great, we got the flag - `cryptanalysis_is_hard`