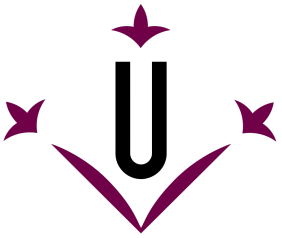

**Universitat de Lleida**

Escola Politècnica Superior
Grau en Enginyeria Informàtica
Algorítmica i Complexitat
Docent: Jordi Planes Cid

# PRÀCTICA 1

Nom: Àiax Faura Vilalta i Joan Martí Olivart
Data entrega: 07/04/2019

# Algorisme de detecció

Per l'algorisme de detecció s'ha optat per un checksum.

## Al fitxer senderscribe.py:

Es suma el codi ascii de tots els caràcters i s'afegeix al final del text, separat per un espai. Per tal de reduir la llargada d'aquest i que l'increment del text sigui el mínim possible se li fa mòdul 100. Es podria fer mòdul 37 ja que és el nombre de caràcters admesos pel programa però, com que també s'utilitza per a l'algoritme de correcció, com es veurà més endavant, s'ha necessitat que fos mòdul 100.
Exemple:
TEXT → ord(T) + ord(E) + ord(X)... = 84 + 69 + 88… = 325 → 325 % 100 = 25
Al fitxer sent.txt queda la següent estructura, on Codi de detecció és la suma i codi de localització es veurà en el següent algorisme:
Text Original+" "+Codi de localització+" "+Codi de detecció

Per a separar els codis i el text s'utilitzen espais. Les cometes son per remarcar l'ús d'aquests.

## Al fitxer receiverscribe.py:

Es fa el mateix checksum que al senderscribe.py i el resultat es compara amb el codi de detecció que s'havia guardat al final del fitxer sent.txt. Per tant, si el text s'ha corromput de la següent forma:
TNXT → ord(T) + ord(N) + ord(X)... = 84 + 78 + 88… = 334 → 334 % 100 = 34
34 != 25 → Hi ha un error en el text

# Algorisme de correcció

Per a l'algorisme de correcció s'utilitzen dos algorismes. Un de localització, per saber on està l'error del text, i un de correcció per saber quin era el caràcter que hi anava anteriorment. Per aquest últim s'utilitza el codi de detecció que obtenim de l'algorisme anterior.

Per tant al final tindrem la següent estructura:
Text Original+" "+Codi de localització+" "+Codi de detecció

## Al fitxer senderscribe.py:

**Per a localitzar on està l'error:**
Es fa mòdul 4 del primer caràcter i es passa a binari de manera que queden dos bits. Es fa el mateix per a tots els caràcters i s'afegeix a un string que es dirà codi de localització.
Exemple:

TEXT
En ASCII la T és, 84 → 84 % 4 = 0 → Passat a binari agafant dos dígits serà 00(S'agafen 2 bits per caràcter per a separar-los. Després sabrem que llegint de dos en dos bits obtindrem un caràcter).

Es fa amb tots els caràcters i s'obté: 00010000
Es passa a hexadecimal per a escurçar el text i queda : 10

De manera que ara tenim:
TEXT 10 25

**Per a saber quin era el caràcter correcte:**
S'utilitza el checskum del codi de detecció.

## Al fitxer receiverscribe.py:

**Per a localitzar on està l'error:**
Vitiet.py fa la seva feina i queda:
TNXT 10 25

Es converteix el 10 d'hexadecimal a binari i s'obté : 00010000

L'algorisme de localització llegeix el primer caràcter i li fa el mateix que es feia en l'script senderscribe.py, el resultat el compara amb el codi passat.

En ASCII la T és, 84 → 84 % 4 = 0 → Passat a binari agafant dos dígits serà 00
Es compara 00 amb els dos primers dígits del codi de localització. Com que són iguals continua llegint.
En ASCII la N és, 78 → 78 % 4 = 2 → Passat a binari agafant dos dígits serà 10
Es compara 10 amb els dos següents dígits del codi de localització. 01 != 10 per tant l'error hi ha un error.
Es retorna la posició anterior en aquest cas 1.

**Per a saber quin era el caràcter correcte:**
Un cop es sap on està l'error s'agafa el caràcter incorrecte que es vol arreglar. noisy_text(pos)? → N (per aquest exemple)

Es mira la diferència entre el codi de detecció (checksum) calculat al receiverscribe.py i el codi enviat des del senderscribe.py.
detection_code → 25
checker → 34
difference = detection_code - checker = 25 - 34 = -9

Aquesta diferència es suma amb el codi ascii del caràcter erroni:
ord(N) + (-9) = 78 - 9 = 69

El caràcter correcte és el que pertany a aquest nombre en la taula ASCII:
char(69) = E

Per tant, hi ha un error a la posició 1 i el caràcter correcte és E.

**Aclariments:**
Aquest algorisme es pot adaptar segons la necessitat que es tingui. Si es vol un increment en el text baix, s'utilitza un mòdul igual a 2 o 4, amb l'inconvenient que com més baix és aquest, més probabilitats de fallada hi haurà en la correcció del codi. Si es prefereix tenir més fiabilitat, es canvia el mòdul a 8, 16 o un nombre més gran tenint en compte que com més gran sigui, més gran serà l'increment en el text.
S'ha adaptat el codi per a que canviar el mòdul sigui tant fàcil com canviar la variable mòdul dels dos scripts principals i que la resta de paràmetres s'adaptessin.
Per exemple, per a especificar la llargada de bits que tindrà cada caràcter en el codi de localització, s'ha utilitzat el següent codi en comptes de ficar directament 2, per a mòdul 4 o 3 per a mòdul 8:
*int(math.log2(module))*

S'ha interpretat l'enunciat com que s'ha detectar i corregir errors del text original, no del codi afegit. Però el make file checker no diferencia entre el text original i codi de detecció-correció i si en aquest últim troba un error, en el make test diu que no és correcte. S'ha considerat un error del make file.

L'increment en el text per a mòdul 4(l'utilitzat en el codi) és poc més de 1.5 i supera els percentatges de detecció i localització especificats en el text.
La probabilitat d'encert de detecció és molt alta, només falla en cas de que es modifiqui un dels espais separadors de codis.
La probabilitat d'encert de correcció és del 75%, ja que ¼ dels caràcters tindrà el mateix mòdul i no detectarà la posició on hi ha error, però a la resta si.

Per a mòdul 8 és de 1.75 i ofereix uns millors resultats.
A partir de mòdul 16, com que l'increment és poc més de 2, seria millor copiar el text al final i afegir un checksum per a comparar-los. Amb això s'obtindria una detecció i correció del 100% i amb menys costos teòrics.

# Canvis en el funcionament després de programar-ho

Si es s'observa el text creat per l'script senderscribe.py es veu que el codi de localització està dividit en dues parts.

1USKJGXOMXKF 1M6O3 C8JNGEQ P82HVQH K38MIHQ5UKA K0BDIMJR9U
JY6S7GQLWAS A1CB8PQ Z2OK6E VMJUKGZZYDJ RE55AG00 7Z5 B2 4
LUYJ50BN998G 6Z7 O4ETJ 3OAU7472O DPB8Z3IX ZZ3 NXYAALDKB1 W5IHMO

0TLFCY65P0 P4DU0CS JRCVYZI UHGVA8U8AXX TGTU4DF XLQU7YFLF73
M2NEJ2EBDIN M5T76N2R44Q HGI36FA8SE J5KFEMZ6 0TONLSGCE LW73 IYQ11Y
68N4RC H14IIVK1K 7C573 WQ8PMJUS0CZ ECSM496C 01
7ecd385bccad40890f14574c85a526fd3717812be499fa48957039280164a532b3123d73b02
2d0ac85439351c02d90004f2b991391100c4201762f1a698614ea810de9349e5a038fd0fc555
222c116dcf7cd067387d1b 85

Abans del codi hexadecimal del codi de localització hi apareixen de 0 a 3 bits, que son el principi del codi de localització.
S'han separat aquests bits pel següent motiu:
Si es té el codi de localització: 011111
Al convertirlo a hexadecimal queda de la següent manera: 1F
Al tornar-ho a convertir a binari en l'script receiverscribe.py queda: 00011111
S'han zeros de manera que al aplicar l'algorisme no funcionarà correctament.

Per això al principi es mira la llargada del text en binari, se li aplica mòdul 4(cada caràcter hexadecimal te 4 bits) i el resultat és el nombre de bits que s'hauran de ficar separats al principi. Amb l'exemple anterior 011111:
len(011111) = 6 → 6 % 4 = 2
Es separen els dos primers bits i la resta del codi es converteix a hexadecimal: 01 F

# Codi algorismes principals:

**Algorisme iteratiu:**

```
def location_algorithm(bin_location_code, noisy_text, module):
        # n és el nombre de caràcters que llegirà cada cop del codi de localització
        n = int(math.log2(module))
        pos = 0

        while bin_location_code != '':
                checker = ord(noisy_text[pos]) % module
                # int(bin_location_code[:n], 2) llegeig els n primers caràcters del location code
i els passa a enter
                if checker != int(bin_location_code[:n], 2):
                        return pos
                # Elimina els caràcters  llegits del codi
                bin_location_code = bin_location_code[n:]
                pos += 1

        return -1
```

**Algorisme recursiu**:

```
  def recursive_loc_algorithm(bin_location_code, noisy_text, module, pos, n):
        if bin_location_code == '':
        return -1
```

```
checker = ord(noisy_text[pos]) % module
if checker != int(bin_location_code[:n], 2):
return pos

return recursive_loc_algorithm(bin_location_code[n:], noisy_text, module, pos + 1, n)
```

# Costos teòrics

Essent n la llargada del text original:
El cost de l'algorisme de detecció  tant iteratiu com recursiu es de **O(n)**, ja que llegeix tot el text un cop.
El cost màxim de l'algorisme de correció tant iteratiu com recursiu es de **O(n)**, ja que llegeix un text de llargada 2n de dos en dos, i el cost de saber quin era el caràcter original és O(1).

# Costos empírics

S'ha mirat els costos 4 cops i s'ha fet la mitjana:

| Costos 1 | Costos 2 | Costos 3 | Costos 4 | Mitjana |
|---|---|---|---|---|
| 2000 0.25<br>4000 0.22<br>6000 0.28<br>8000 0.32<br>10000 0.39 | 2000 0.21<br>4000 0.22<br>6000 0.30<br>8000 0.35<br>10000 0.38 | 2000 0.19<br>4000 0.24<br>6000 0.28<br>8000 0.35<br>10000 0.39 | 2000 0.18<br>4000 0.25<br>6000 0.29<br>8000 0.33<br>10000 0.39 | 2000 0.21<br>4000 0.23<br>6000 0.29<br>8000 0.34<br>10000 0.39 |

# Aclariments del codi

El fitxer checker.py s'ha editat perquè el codi de localització conté minúscules, i al fer el make test no funcionava. Per tant a la línia 21 del fitxer checker.py s'han afegit les lletres minúscules al domain. Si no es considera adequat aquest canvi, el codi mostrat en hexadecimal amb lletres minúscules es podria mostrar en lletres majúscules, però complicava una mica més el codi i per això s'ha optat per aquest camí.

Al senderscribe.py a la següent línia de codi:
*hex(int(location_code, 2))[2:].zfill(int(len(location_code)/4))*

El codi en negre passa el codi de localització de binari a hexadecimal. La part en blau s'utilitza pel següent motiu:

Si es té 0000000000000011 → 14 zeros i 2 uns, al convertir-ho a hexadecimal python elimina els 0's del principi de manera que quedarà que 0000000000000011 en hexadecimal és 3. I com que els zeros son rellevants per l'algorisme, el codi en blau ompla els zeros que faltin per a que la informació sigui correcta. LLavors queda: 0003 en hexadecimal

Pero al receiverscribe.py, al llegir-ho ens trobem amb un problema semblant. Quan python passa el 0003 a hexadecimal esborra els zeros del principi de manera que només queda: 11. Per això s'utilitza el següent codi que afegeix els zeros que falten:
*bin_location_code = bin(int(location_code, 16))[2:].zfill(len(location_code) * 4)*