Mystère, chargement et sélection

INFORMATIQUE COMMUNE - Devoir n° 2 - Olivier Reynet

Consignes :

1. utiliser une copie différente pour chaque partie du sujet,
2. écrire son nom sur chaque copie,
3. écrire de manière lisible et intelligible,
4. préparer une réponse au brouillon avant de la reporter sur la feuille.

A Notions de base

A1. Un code utilise un tableau de 100 000 entiers codés sur 16 bits. Donner la taille de l'espace mémoire nécessaire pour stocker ce tableau en Mo.

Solution : On a besoin de $N \times (16/8) = N$ octets soit 0,2 Mo.

A2. Un code utilise un tableau dont la dimension est 1 000 lignes et trois colonnes. Chaque ligne est composée de trois nombres codés sur 64 bits. Donner la taille de l'espace mémoire nécessaire pour stocker ce tableau en Ko.

Solution : On a besoin de $N \times 3 \times (64/8) = 24N = 24\,000$ octets soit 24 Ko.

A3. Coder les instructions nécessaires à la création de la liste `[[],[],[],[],[],[],[],[],[],[]]`.

Solution :

```
L = []
for _ in range(10):
    L.append([])
# ou bien
L = [[] for _ in range(10)]  # liste en compréhension
```

A4. Prouver que la fonction mystère ci-dessous se termine.

```
def mystere(L):
    r = ""
    while len(L) > 0:
        e = L.pop()
        r += chr(e)
    return r
```

Solution : `k=len(L)` est un variant de boucle pour la boucle while. k est un entier positif avant la boucle. À chaque tour de boucle, il décroît d'une unité à cause de l'instruction `pop`. D'après le théorème de la limite monotone, il atteint nécessairement zéro au bout d'un certain nombre d'itérations. La condition de boucle est donc invalidée et l'algorithme se termine.

A5. Quel est le résultat de la fonction mystère appliquée à la liste
`[33, 32, 108, 105, 114, 118, 97, 39, 100, 32, 110, 111, 115, 115, 105, 111, 80]`?
Pour répondre à la question, il est nécessaire d'utiliser la documentation en annexe et on rappelle ci-dessous la documentation de la fonction `chr` en Python :

> **chr(i)** Return the string representing a character whose ASCII code point is the integer i. For example, chr(97) returns the string 'a', while chr(8364) returns the string ''. This is the inverse of ord().

Solution : Poisson d'avril!

A6. On considère la suite $(u_n)_{n\in\mathbb{N}}$ définie par $u_n = 2u_{n-1} + 1$ et u_0 est un nombre entier connu. Coder une fonction de prototype `un(u0, n)` qui renvoie le nième terme de la suite $(u_n)_{n\in\mathbb{N}}$. On donnera deux versions de cette fonction : l'une itérative et l'autre récursive.

Solution :

```
def rec_un(u0,n):
    if n==0:
        return u0 # condition d'arrêt
    else:
        return 2*un(u0, n-1) + 1  # appel récursif

def ite_un(u0, n):
    u = u0
    while n > 0:
        u = 2*u +1
        n -= 1
    return u
```

A7. On souhaite rechercher par dichotomie un élément dans le tableau `[1,3,6,7,11,15,18,21,33,42,99]`. Détailler les étapes de la recherche de l'élément 21 sur le tableau.

Solution : Exemple : on recherche 21 dans le tableau :
`[1,3,6,7,11,15,18,21,33,42,99]`
indice du milieu = 5 -> on recherche au dessus -> `[18,21,33,42,99]`
indice du milieu = 8 -> on recherche au dessous `[18,21]`
indice du milieu = 6 -> on recherche au dessus `[18,21]`
indice du milieu = 7 -> trouvé

A8. Coder en Python la fonction `rdicho(t,e)` qui renvoie l'indice de l'élément si celui-ci est contenu dans le tableau et `None` sinon.

Solution :

```
def dichotomic_search(t, elem):
    g = 0
    d = len(t) - 1
    while g <= d:
        m = (d + g) // 2
        # print(g, m, d)
        if t[m] < elem:
            g = m + 1
        elif t[m] > elem:
            d = m - 1
        else:
            return m
    return None
```

B Chargement d'un conteneur

Vous disposez d'un conteneur dont le volume maximum est V_{max} m^3 et de caisses. Les poids p en kg et le volume v en m^3 de chaque type de caisse sont connus. On dispose d'autant de caisses que l'on veut pour chaque type de caisse. Votre objectif est de maximiser le poids total embarqué dans le conteneur tout en respectant le volume maximal admissible par le conteneur (V_{max}). Vous pouvez composer le chargement comme vous le souhaitez en ce qui concerne le nombre de chaque type de caisse.

On dispose d'une liste `C=[(140, 3), (220, 5), (160, 4), (30, 1), (50, 2)]` répertoriant les poids et volumes des caisses : `C[i][0]` contient le poids de la caisse et `C[i][1]` le volume de la caisse.

B1. Proposer deux stratégies gloutonnes différentes pour tenter de trouver une solution à ce problème.

Solution : On peut soit :

S1 choisir la caisse de poids maximal en premier.

S2 choisir la caisse dont le rapport poids/volume est maximal en premier.

B2. Appliquer à la main ces deux stratégies gloutonnes pour des volumes de conteneur `V1 = 5` m^3 et `V2 = 7` m^3 et les caisses `C=[(140, 3), (220, 5), (160, 4), (30, 1), (50, 2)]`.

Solution : Pour `V1=5` m^3

S1 On trie la liste en fonction du poids `[(220, 5), (160, 4), (140, 3), (50, 2), (30, 1)]`. On choisit le plus élevé en premier. On obtient `S= [(220,5,1)]`, soit une seule caisse de 5 m^3 et 220 kg.

S2 On trie la liste en fonction du rapport poids/volume `[(46.7, 140, 3), (44., 220, 5), (40., 160, 4), (30., 30, 1), (25., 50, 2)]`. On choisit le rapport le plus élevé en premier. On obtient `S= [(140, 3, 1), (30, 1, 2)]`, soit une caisse 3 m^3 et deux de 1 m^3. Le poids est de $140 + 30 \times 2 = 200$ kg.

Pour `V2=7` m^3 on trouve :

S1 [(220, 5, 1), (50, 2, 1)] et on obtient 270 kg.

S2 [(140, 3, 2), (30, 1, 1)] et on obtient 310 kg.

B3. Comparer les résultats obtenus. Ces stratégies gloutonnes sont-elles optimales ?

Solution : Aucune de ces stratégies gloutonnes n'est optimale, puisque l'une trouve toujours mieux que l'autre dans les exemples précédents et ce n'est pas la même qui trouve le meilleur résultat dans les deux cas.

B4. Choisir une des deux stratégies et coder une fonction de prototype `greeedy_load(C, Vmax)` qui l'implémente. Préciser la stratégie choisie. On supposera que la liste `C` est triée correctement, selon les besoins de la stratégie.

Solution :

```
def greedy_load(C, Vmax):
    ratios = sorted([(v / l, v, l) for v, l in C])
    print(ratios)
    remaining_vol = Vmax
    total_weight = 0
    S = []  # solution
    i = len(ratios) - 1
    while i >= 0  and remaining_vol > 0:
        ratio, weight, vol = ratios[i]  # choose the best ratio, the last
        n = remaining_vol // vol  # how many times ?
        if n > 0 and vol <= remaining_vol:  # is it a solution ?
            S.append((weight, vol, n))
            total_weight += n * weight
            remaining_vol -= n * vol
        i -= 1
    return S, remaining_vol, total_weight

def v_greedy_load(C, Vmax):
    objets = sorted([(v, l) for v, l in C])
    print(objets)
    remaining_vol = Vmax
    total_weight = 0
    S = []  # solution
    i = len(objets) - 1
    while  i >= 0 and remaining_vol > 0:
        weight, length = objets[i]  # choose the best ratio, the last
        n = remaining_vol // length  # how many times ?
        if n > 0 and length <= remaining_vol:  # is it a solution ?
            S.append((weight, length, n))
            total_weight += n * weight
            remaining_vol -= n * length
        i -= 1
    return S, remaining_vol, total_weight
```

```
#MAIN PROGRAM
C = [(140, 3), (220, 5), (160, 4), (30, 1), (50, 2)]
Vmax = 5

gc = v_greedy_load(C, Vmax)
print("V", Vmax, gc)
gc = greedy_load(C, Vmax)
print("R", Vmax, gc)

Vmax = 7

gc = v_greedy_load(C, Vmax)
print("V", Vmax, gc)
gc = greedy_load(C, Vmax)
print("R", Vmax, gc)
```

C Trouver les deux points les plus proches

Soit un ensemble de n points ($n \geqslant 3$) du plan. On utilise un repère orthonormé et les points sont représentés par leurs coordonnées cartésiennes (cf. figure 1). Pour mesurer la distance entre deux points, on utilise la distance euclidienne. Pour deux points $P_i(x_i, y_i)$ et $P_j(x_j, y_j)$, la distance vaut $d(P_i, P_j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$.

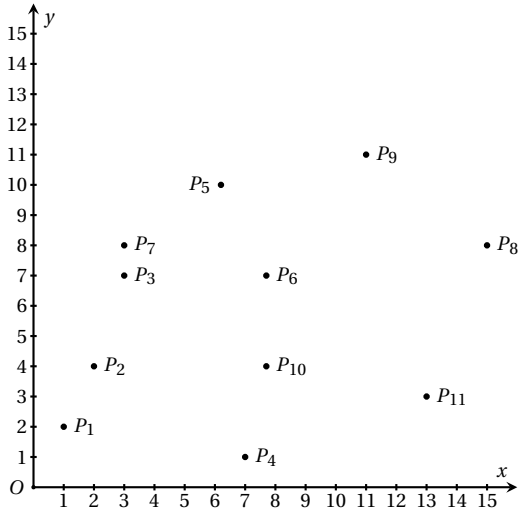

FIGURE 1 – Exemple d'ensemble de points : on cherche les deux points les plus proches dans un nuage de points.

On représente les points en Python par des tuples `(x_i, y_i)`. On dispose de la liste `P` des points de l'ensemble :

```
P = [(1, 2),(2, 4),(3, 7),(7, 1),(6.2, 10),(7.7, 7),(3, 8),(14, 8),(11, 11),(7.7, 4),(12, 3)]
```

C1. Écrire une fonction de prototype `distance(p, q)` qui renvoie la distance euclidienne entre deux points `p` et `q`. On prendra soin d'importer les bibliothèques nécessaires.

Solution :

```
import math

def distance(p, q):
    return math.sqrt((p[0] - q[0]) ** 2 + (p[1] - q[1]) ** 2)
```

C2. En calculant toutes les distances entre toutes les paires de points, écrire une fonction Python de prototype `naif_ppproche(L)` qui renvoie un tuple constitué par :

1. la distance minimale entre deux points,
2. et la paire de points ainsi constituée sous la forme d'une liste de deux tuples.

Par exemple, pour la liste de points de la figure 1, la fonction renvoie `(1.0, [(3, 7), (3, 8)])`. Pour initialiser la distance minimale, il est possible d'utiliser `math.inf` qui représente l'infini, mais ce n'est pas la seule solution.

Solution :

```
def naif_ppproche(P):
    dmin = math.inf
    paire = None
    for i in range(len(P)):
        p = P[i]
        for j in range(i+1,len(P)):
            q = P[j]
            d = distance(p, q)
            if d < dmin:
                paire = [p, q]
                dmin = d
    return dmin, paire
```

C3. Quelle est la complexité temporelle de la fonction `naif_ppproche` ? On prendra soin de bien préciser de quel(s) paramètre(s) dépend la complexité et de calculer précisément la complexité.

Solution : La complexité temporelle de `naif_pppaire` dépend de la taille de la liste de points `P` . Notons n la taille de cette liste. Dans tous les cas, si n est fixé, les nombres d'itération des boucles seront les mêmes. Il n'y donc pas de pire ni de meilleur des cas. On fait l'hypothèse que les instructions à l'intérieur des boucles sont effectuées en un temps constant c par la machine.

$$C(n) = c + \sum_{i=0}^{n-1} \sum_{j=i+1}^{n-1} c = c + c \sum_{i=0}^{n-1} (n-1-i-1+1) = c + c \sum_{i=0}^{n-1} (n-1-i)$$

$$C(n) = c + c(n-1)n - c\frac{(n-1)n}{2} = c(1 + \frac{(n-1)n}{2}) = O(n^2)$$

La complexité de cette approche naïve est quadratique.

On souhaite **améliorer** cette complexité en implémentant l'algorithme **PPPROCHE** qui procède comme suit :

1. Si le nuage de points comporte trois points ou moins, on applique l'algorithme naïf.
2. Sinon, on divise le plan en deux parties G et R selon l'axe des abscisses. L'axe choisi est la droite verticale dont l'abscisse est celle du point médian selon l'axe (Ox) (cf droite Δ sur la figure 2)). Ensuite, on effectue les étapes suivantes :
   (a) résolution du problème récursivement dans la partie G et dans la partie R
   (b) sélection de la paire la plus proche de G et de R : (P_m, P_n) est a une distance d_{min}.
   (c) construction de la bande intermédiaires de largeur $2d_{min}$ et recherche d'une paire de points (P_g, P_r) à cheval sur G et R plus proche que d_{min} dans cette bande.
   (d) renvoyer la paire qui présente la distance la plus petite entre (P_m, P_n) et (P_g, P_r) (si elle existe).

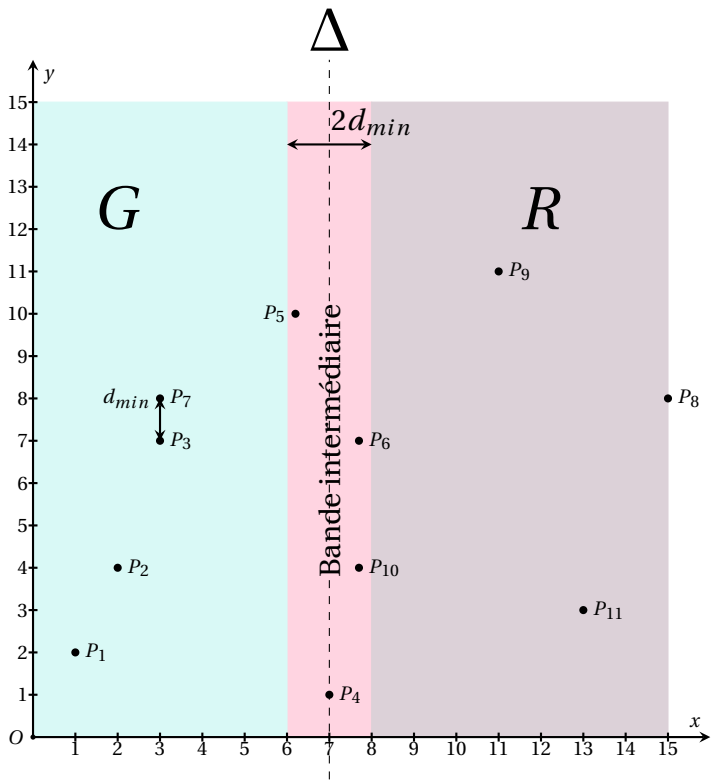

FIGURE 2 – L'espace de recherche a été divisé en deux partie G et R par rapport à la droite Δ dont l'abscisse est celle du point médian selon l'axe (Ox). On a trouvé dans la partie G la paire de points la plus proche (P_3, P_7) : ces points sont espacés de d_{min}. On construit alors la bande intermédiaire centrale qui s'étend sur une largeur de $2d_{min}$ autour de Δ. On recherche dans cette bande une éventuelle paire de points plus proche que d_{min}, c'est-à-dire une paire de points à cheval sur G et R.

C4. Pour implémenter cet algorithme, il est nécessaire de trier la liste des points du plan selon l'abscisse des points ou selon l'ordonnée des points. On choisit d'implémenter l'algorithme de tri fusion de manière **récursive**. On dispose déjà d'une méthode `fusion(t1, t2, axe)` de complexité linéaire qui permet de fusionner deux tableaux de points triés selon le paramètre entier `axe` (0 pour les abscisses

et 1 pour les ordonnées). Écrire une fonction `tri_fusion(t, axe)` qui implémente le tri d'une liste de points selon un axe donné.

Solution :

```
def divise_en_2(t):
    n = len(t)
    t1, t2 = [], []
    for i in range(n // 2):
        t1.append(t[i])
    for i in range(n // 2, n):
        t2.append(t[i])
    # Slicing
    # t1 = t[0:n // 2]
    # t2 = t[n // 2:n]
    return t1, t2


def fusion(t1,t2,axe):
    n1 = len(t1)
    n2 = len(t2)
    if n1 == 0:
        return t2
    elif n2 == 0:
        return t1
    elif  t1[0][axe] <= t2[0][axe]:
        return [t1[0]] + fusion(t1[1:], t2, axe)
    else:
        return [t2[0]] + fusion(t1, t2[1:], axe)


def tri_fusion(t, axe):
    if len(t) < 2:
        return t
    else:
        t1, t2 = divise_en_2(t)
        return fusion(tri_fusion(t1, axe), tri_fusion(t2, axe), axe)
```

C5. Quelle est la complexité de la fonction `tri_fusion` ? Expliquer brièvement pourquoi sans faire de calculs.

Solution : $O(n \log n)$ cf. cours

C6. Écrire une fonction de prototype `select_bande(P, x_delta, dmin)` qui renvoie la liste des points qui appartiennent à la bande du milieu, c'est-à-dire la bande du plan centrée en l'abscisse `x_delta` et de largeur `2*dmin` (cf. figure 2).

Solution :

```
def select_bande(P, x_delta, dmin):
    bande = []  # liste des points dans la bande
    for i in range(len(P)):
```

```
            if abs(P[i][0] - x_delta) < dmin:
                bande.append(P[i])
        return bande
```

En plus des fonctions `merge_sort` et `select_bande`, on dispose d'une fonction de prototype `ppproche_bande(bande, dmin)` de complexité $O(n \log n)$ qui trouve, si elle existe, la paire de points la plus proche dans la bande intermédiaire. Cette fonction renvoie `(dmin, (P_g,P_r))` et, si cette paire n'existe pas, `(dmin, None)`.

C7. Compléter le code ci-dessous afin d'implémenter l'algorithme PPPROCHE décrit plus haut.

```
def ppproche(P):
    n = len(P)
    # à compléter !
    # cette fonction est récursive
    # elle renvoie le tuple ---> dmin, paire_la_plus_proche

def paire(P):
    lst = merge_sort(P, 0) # P est triée selon l'axe des x dans le sens croissant
    return ppproche(lst)

# Programme principal
P = [(1, 2),(2, 4),(3, 7),(7, 1),(6.2, 10),(7.7, 7),(3, 8),(14, 8),(11, 11),(7.7, 4),(12, 3)]
dmin, paire = paire(P)
```

Solution :

```
def ppproche_bande(bande, dmin):
    # on peut faire mieux que cette implémentation !
    paire = None
    b = tri_fusion(bande, 1)
    for i in range(len(bande)):
        for j in range(i + 1, len(bande)):
            if distance(b[i], b[j]) < dmin:
                dmin = distance(b[i], b[j])
                paire = [b[i], b[j]]
    return dmin, paire

def ppproche(P):
    n = len(P)
    print("Call with n=", n)
    if n <= 3:
        return naif_ppproche(P)
    else:
        i_med = n // 2
        p_xmed = P[i_med]
        dl, pl = ppproche(P[:i_med])  # distance min à gauche
        dr, pr = ppproche(P[i_med:])  # distance min à droite
        d = dl if dl < dr else dr
        p = pl if dl == d else pr
        bande = select_bande(P, p_xmed[0], d)
        db, pb = ppproche_bande(bande, d)
        if db < d:
            return db, pb
        else:
            return d, p


def paire(P):
    lst = tri_fusion(P, 0)
    return ppproche(lst)


# Programme principal
P=[(1,2),(2,4),(3,7),(7,1),(6.2,10),(7.7,7),(3,8),(14,8),(11,11),(7.7,4),(12,3)]
```

```
    dmin, paire = paire(P)
```

C8. Justifier que la complexité temporelle de cet algorithme s'exprime par la relation de récurrence : $T(n) = n\log n + 2T(n/2)$. En supposant que la liste de point comporte $n = 2^k$ points, déduire que $T(n) < nT(1) + n(\log n)^2$. Conclure.

Solution : On calcule la complexité de la fonction `ppproche` en comptant le tri fusion initial.

$$T(n) = n\log n + 2T(n/2) + n\log n = 2T(n/2) + n\log n < 2^k T(n/2^k) + kn\log n < nT(1) + n(\log n)^2$$

Le complexité de l'algorithme est donc en $O(n(\log n)^2)$.

D Annexe

ASCII TABLE

Decimal	Hexadecimal	Binary	Octal	Char
000	00	0000000	000	*[NULL]*
001	01	0000001	001	*[START OF HEADING]*
002	02	0000010	002	*[START OF TEXT]*
003	03	0000011	003	*[END OF TEXT]*
004	04	0000100	004	*[END OF TRANSMISSION]*
005	05	0000101	005	*[ENQUIRY]*
006	06	0000110	006	*[ACKNOWLEDGE]*
007	07	0000111	007	*[BELL]*
008	08	0001000	010	*[BACKSPACE]*
009	09	0001001	011	*[HORIZONTAL TAB]*
010	0A	0001010	012	*[LINE FEED]*
011	0B	0001011	013	*[VERTICAL TAB]*
012	0C	0001100	014	*[FORM FEED]*
013	0D	0001101	015	*[CARRIAGE RETURN]*
014	0E	0001110	016	*[SHIFT OUT]*
015	0F	0001111	017	*[SHIFT IN]*
016	10	0010000	020	*[DATA LINK ESCAPE]*
017	11	0010001	021	*[DEVICE CONTROL 1]*
018	12	0010010	022	*[DEVICE CONTROL 2]*
019	13	0010011	023	*[DEVICE CONTROL 3]*
020	14	0010100	024	*[DEVICE CONTROL 4]*
021	15	0010101	025	*[NEGATIVE ACKNOWLEDGE]*
022	16	0010110	026	*[SYNCHRONOUS IDLE]*
023	17	0010111	027	*[ENG OF TRANS. BLOCK]*
024	18	0011000	030	*[CANCEL]*
025	19	0011001	031	*[END OF MEDIUM]*
026	1A	0011010	032	*[SUBSTITUTE]*
027	1B	0011011	033	*[ESCAPE]*
028	1C	0011100	034	*[FILE SEPARATOR]*
029	1D	0011101	035	*[GROUP SEPARATOR]*
030	1E	0011110	036	*[RECORD SEPARATOR]*
031	1F	0011111	037	*[UNIT SEPARATOR]*
032	20	0100000	040	*[SPACE]*
033	21	0100001	041	**!**
034	22	0100010	042	**"**
035	23	0100011	043	**#**
036	24	0100100	044	**$**
037	25	0100101	045	**%**
038	26	0100110	046	**&**
039	27	0100111	047	**'**
040	28	0101000	050	**(**
041	29	0101001	051	**)**
042	2A	0101010	052	*****
043	2B	0101011	053	**+**
044	2C	0101100	054	**,**
045	2D	0101101	055	**-**
046	2E	0101110	056	**.**
047	2F	0101111	057	**/**
048	30	0110000	060	**0**
049	31	0110001	061	**1**
050	32	0110010	062	**2**
051	33	0110011	063	**3**
052	34	0110100	064	**4**
053	35	0110101	065	**5**
054	36	0110110	066	**6**
055	37	0110111	067	**7**
056	38	0111000	070	**8**
057	39	0111001	071	**9**
058	3A	0111010	072	**:**
059	3B	0111011	073	**;**
060	3C	0111100	074	**<**
061	3D	0111101	075	**=**
062	3E	0111110	076	**>**
063	3F	0111111	077	**?**
064	40	1000000	100	**@**
065	41	1000001	101	**A**
066	42	1000010	102	**B**
067	43	1000011	103	**C**
068	44	1000100	104	**D**
069	45	1000101	105	**E**
070	46	1000110	106	**F**
071	47	1000111	107	**G**
072	48	1001000	110	**H**
073	49	1001001	111	**I**
074	4A	1001010	112	**J**
075	4B	1001011	113	**K**
076	4C	1001100	114	**L**
077	4D	1001101	115	**M**
078	4E	1001110	116	**N**
079	4F	1001111	117	**O**
080	50	1010000	120	**P**
081	51	1010001	121	**Q**
082	52	1010010	122	**R**
083	53	1010011	123	**S**
084	54	1010100	124	**T**
085	55	1010101	125	**U**
086	56	1010110	126	**V**
087	57	1010111	127	**W**
088	58	1011000	130	**X**
089	59	1011001	131	**Y**
090	5A	1011010	132	**Z**
091	5B	1011011	133	**[**
092	5C	1011100	134	****
093	5D	1011101	135	**]**
094	5E	1011110	136	**^**
095	5F	1011111	137	**_**
096	60	1100000	140	**`**
097	61	1100001	141	**a**
098	62	1100010	142	**b**
099	63	1100011	143	**c**
100	64	1100100	144	**d**
101	65	1100101	145	**e**
102	66	1100110	146	**f**
103	67	1100111	147	**g**
104	68	1101000	150	**h**
105	69	1101001	151	**i**
106	6A	1101010	152	**j**
107	6B	1101011	153	**k**
108	6C	1101100	154	**l**
109	6D	1101101	155	**m**
110	6E	1101110	156	**n**
111	6F	1101111	157	**o**
112	70	1110000	160	**p**
113	71	1110001	161	**q**
114	72	1110010	162	**r**
115	73	1110011	163	**s**
116	74	1110100	164	**t**
117	75	1110101	165	**u**
118	76	1110110	166	**v**
119	77	1110111	167	**w**
120	78	1111000	170	**x**
121	79	1111001	171	**y**
122	7A	1111010	172	**z**
123	7B	1111011	173	**{**
124	7C	1111100	174	**|**
125	7D	1111101	175	**}**
126	7E	1111110	176	**~**
127	7F	1111111	177	*[DEL]*