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.

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.

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

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

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

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().

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.

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.

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.

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.

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)]`.

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

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.

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 $d(P_i, P_j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$.

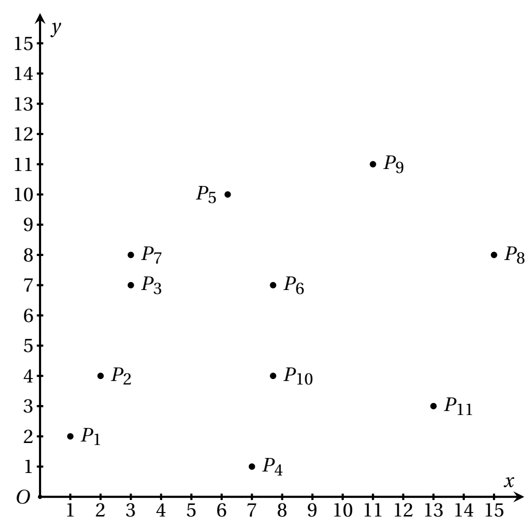

FIGURE 1 – Exemple d'ensemble de points : on cherche les deux points les plus proches dans un nuage de points.

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.

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.

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.

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).

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.

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

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).

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)`.

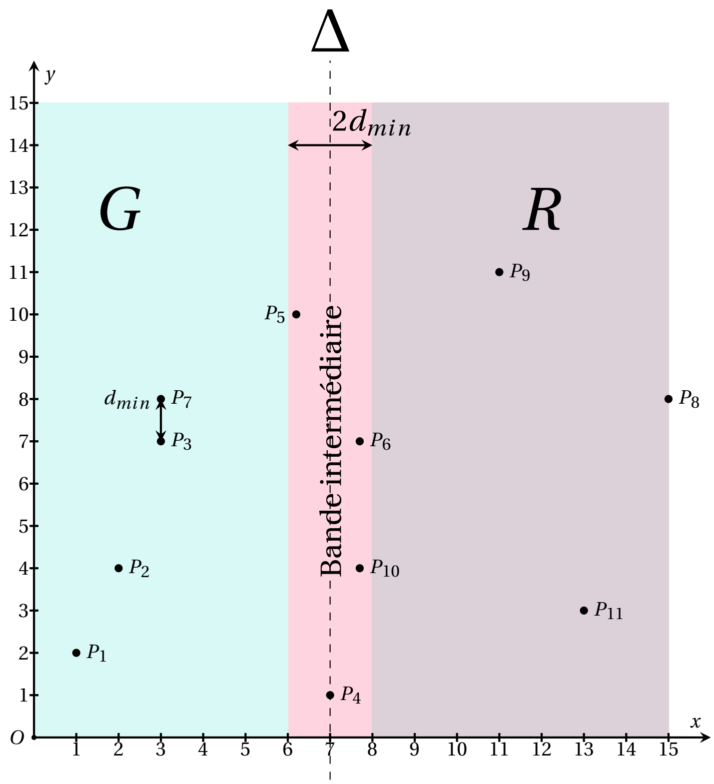

FIGURE 2 – L'espace de recherche a été divisé en deux partie G et R par rapport à la droite Δ dont l'abscisse est celle du point médian selon l'axe (Ox). On a trouvé dans la partie G la paire de points la plus proche (P_3, P_7) : ces points sont espacés de d_{min}. On construit alors la bande intermédiaire 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.

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

```python
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)
```

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.

D Annexe

ASCII TABLE

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