Transport et proximité

INFORMATIQUE COMMUNE - Devoir n° 3 - 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 Intégration et transport

Le Bureau Des Élèves (BDE) d'une école d'ingénieurs organise un séjour d'intégration de trois jours pour les nouvelles recrues. L'école et ses sponsors subventionnent le coût du transport de ce séjour à hauteur de 3000 € **maximum**.

L'équipe du BDE en charge du transport souhaite véhiculer **un maximum d'élèves tout en respectant ce budget.** Ils ont identifié plusieurs moyens de transports possibles :

1. la location de bus, par groupe de 51,
2. la location de mini-bus, par groupe de 7,
3. le train, par groupe de 10,
4. le covoiturage, par groupe de 3,

Le tableau 1 fait la synthèse des capacités et des prix de ces différentes solutions.

Moyen de transport	**Bus**	**Mini-bus**	**Train**	**Covoiturage**
Nombre d'élèves transportés	51	7	10	3
Prix (€)pour ce nombre d'élèves	510	91	200	45

TABLE 1 – Informations sur les moyens de transports envisagés.

A1. Proposer une stratégie gloutonne pour trouver une solution à ce problème. On suppose qu'on dispose d'autant de bus, de train, de mini-bus et de covoiturage que nécessaire.

En Python, les données du tableau 1 sont représentées sous la forme d'une liste de tuples :
`T = [(51, 510), (7, 91), (3, 45), (10, 200)].`
On rappelle que `T[i][0]` permet donc d'accéder à la capacité et `T[i][1]` au prix du moyen de transport.

A2. Appliquer à la main cette stratégie gloutonne pour un budget maximum de 3000 € et un nombre d'élèves à transporter de 300.

A3. On suppose que `prix` représente le prix à payer pour utiliser un moyen de transport τ et que `B` est le budget disponible. Comment peut-on calculer :

— `n`, le nombre de fois que l'on peut utiliser τ ?

— `b`, le budget restant après utilisation de `n` fois τ ?

Donner les instructions Python nécessaires à ces calculs.

A4. On souhaite appliquer la stratégie gloutonne. Dans ce but, il est nécessaire de trier une liste de tuple d'après le **premier élément de chaque tuple** dans l'ordre décroissant. Compléter les lignes du code suivant qui est un tri en indiquant les numéros de la ligne à laquelle vous faites référence (`# LIGNE x`). De quel tri s'agit-il ?

```python
def h(t1, t2):
    n1 = len(t1)
    n2 = len(t2)
    if n1 == 0:
        # LIGNE 1
    elif n2 == 0:
        # LIGNE 2
    else:
        if # LIGNE 3:
            return [t1[0]] + h(t1[1:], t2)
        else:
            return  # LIGNE 4


def g(t):
    n = len(t)
    if n < 2:
        # LIGNE 5
    else:
        t1, t2 = # LIGNE 6
        return h(g(t1), g(t2))
```

A5. Coder une fonction de prototype `repartition(T, N, Bmax)` qui implémente la stratégie gloutonne pour transporter un maximum d'élèves parmi les `N` élèves, tout en respectant le budget maximum `Bmax`. Cette fonction renvoie la solution sous la forme d'une liste de tuples `(capacite, prix, n)`, où `n` est le nombre de fois qu'on a utilisé le transport.

Par exemple, `repartition(T, 300, 3000)` renvoie `[(51, 510, 5), (10, 200, 2), (3, 45, 1)]`, selon la stratégie choisie.

A6. Écrire une fonction de signature `S_vers_ne_p(S)` dont le paramètre d'entrée est la sortie de la fonction précédente et qui renvoie le tuple `ne, p`, où `ne` est le nombre d'élèves transportés et `p` le prix total payé.

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

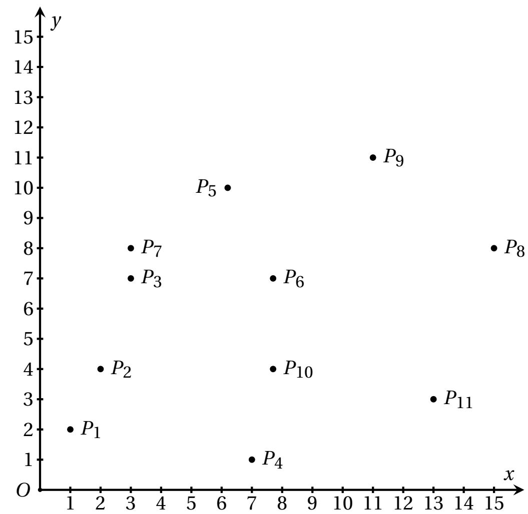

FIGURE 1 – Exemple d'ensemble de points : on cherche les deux points les plus proches dans un nuage de points.

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)]
```

B7. 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 si nécessaires.

B8. 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.

B9. Quelle est la complexité temporelle de la fonction `naif_ppproche` ? On prendra soin de bien de 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).

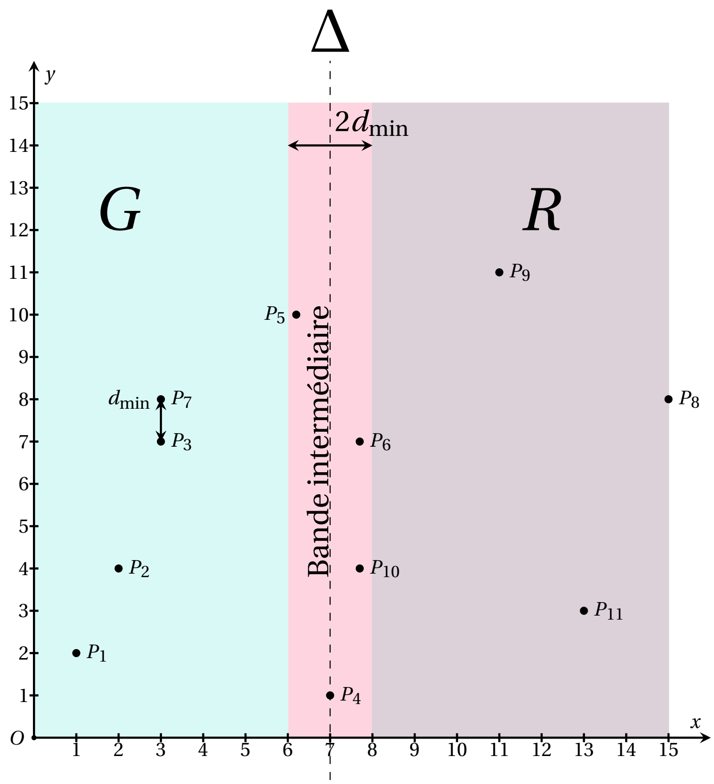

FIGURE 2 – L'espace de recherche a été divisé en deux partie G et R par rapport à la droite Δ dont l'abscisse est celle du point médian de l'ensemble des points selon l'axe (Ox), P_4 dans cet exemple. On a trouvé dans la partie G la paire de points la plus proche (P_3, P_7) : ces points sont espacés de d_{min}. On construit alors la bande intermédiaire centrale qui s'étend sur une largeur de $2d_{\text{min}}$ 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 plus proche que d_{min}.

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 un algorithme de tri efficace que vous pouvez utiliser ainsi : `tri(t, axe)`, où `axe` vaut 0 pour trier selon les abscisses et 1 pour trier selon les ordonnées.

B10. Citer un algorithme de tri efficace qui n'est pas celui implémenté dans la première partie de cet examen et donner sa complexité dans pire des cas et le meilleur des cas.

B11. 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).

B12. Écrire une fonction de signature `ppproche_bande(bande, dmin)` qui renvoie, si elle existe, une paire de points plus proche que `dmin` dans la bande intermédiaire (point c. de l'algorithme PPPROCHE décrit plus haut). Cette fonction renvoie `(n_dmin, (P_g,P_r))` et, si cette paire n'existe pas, `(dmin, None)`.

B13. Compléter le code ci-dessous afin d'implémenter l'algorithme PPPROCHE décrit plus haut. Exprimer la fonction `ppproche` de manière **récursive**.

```
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 = tri(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)
```