Des puces, des sauts et des paquets cadeaux

INFORMATIQUE COMMUNE - Devoir n° 4 - 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.

Les trois parties de ce contrôle sont indépendantes.

A Une puce aux nombreuses fonctionnalités (cours)

Une entreprise fabrique des puces électroniques. Ces puces composées de différents circuits, chaque circuit proposant plusieurs fonctionnalités (addition, multiplication, lien série, bus I2C, timer, ADC, DAC, thermomètre, accéléromètre...). **L'entreprise cherche à implanter le plus de fonctionnalités possible sur une surface limitée `smax` de silicium, le support des circuits**.

Chaque circuit est caractérisé par le nombre de fonctionnalités qu'il propose et la surface qu'il occupe sur le silicium.

En Python, on modélise donc la liste des circuits sous la forme d'une liste de tuples :

- le premier élément du tuple représente le nombre de fonctionnalités du circuit,
- le second la surface en mm^2 nécessaire pour graver le circuit sur le silicium.

Par exemple, une liste possible de circuits est : `C = [(10, 1), (15, 5), (13, 2), (12, 2), (7, 4), (5, 3)]`.

A1. Décrire deux stratégies gloutonnes différentes pour résoudre ce problème. Vous préciserez à chaque fois l'ordre glouton utilisé.

> **Solution :** Deux stratégies gloutonnes possibles sont :
>
> **A** choisir le circuit possédant le plus de fonctionnalité en premier. L'ordre glouton est donc de trier cette liste de tuples selon le nombre de fonctionnalités décroissant.
>
> **B** choisir le circuit dont le rapport fonctionnalité/surface est en premier. L'ordre glouton est donc de trier cette liste de tuples selon ce rapport décroissant.

A2. Appliquer à la main ces deux stratégies gloutonnes pour des surfaces maximales de puce `S1 = 4` mm^2 et `S2 = 9` mm^2 et les circuits `C = [(10, 1), (15, 5), (13, 2), (12, 2), (7, 4), (5, 3)]`.

> **Solution :**

1. Pour `S1=4` mm^2
   - **A** On trie la liste en fonction du nombre de fonctionnalités décroissantes : `[(15, 5), (13, 2), (12, 2), (10, 1), (7, 4), (5, 3)]`. On parcourt la liste dans cet ordre. On obtient `[(13, 2), (12, 2)]`, soit une surface de 4 mm^2 et 25 fonctionnalités.
   - **B** On trie la liste en fonction du rapport fonctionnalités/surface `[(10.0, 10, 1), (6.5, 13, 2), (6.0, 12, 2), (3.0, 15, 5), (1.75, 7, 4), (1.67, 5, 3)]`. On parcourt la liste dans cet ordre. On obtient `[(10, 1), (13, 2)]`, soit une surface de 3 mm^2 et 23 fonctionnalités.
2. Pour `S2=9` mm^2 on trouve :
   - **A** `[(15, 5), (13, 2), (12, 2)]` soit 9 mm^2 et 40 fonctionnalités.
   - **B** `[(10, 1), (13, 2), (12, 2), (7, 4)]` soit 9 mm^2 et 42 fonctionnalités.

A3. Conclure sur l'optimalité de ces deux stratégies.

Solution : On observe qu'aucune stratégie n'est optimale, car si l'une d'entre elles l'était, elle trouverait toujours le nombre de fonctionnalités le plus grand. Or, ce n'est pas le cas.

On souhaite trier la liste de tuples `C` selon un ordre glouton. On propose le tri suivant.

```
1  def g(t1, t2):
2      n1, n2 = len(t1), len(t2)
3      if n1 == 0:
4          return ...  # 1 à compléter
5      elif n2 == 0:
6          return ...  # 2 à compléter
7      elif t1[0][0] > t2[0][0]:
8          return ...  # 3 à compléter
9      else:
10         return ...  # 4 à compléter
11
12 def h(t):
13     n = len(t)
14     if len(t) < 2:
15         return t
16     else:
17         t1, t2 =  ... # 5 à compléter
18         return g(h(t1), h(t2))
```

A4. Compléter les lignes manquantes de cet algorithme de tri.

Solution :

```
1  def g(t1, t2):
2      n1, n2 = len(t1), len(t2)
3      if n1 == 0:
4          return t2
5      elif n2 == 0:
6          return t1
7      elif t1[0][0] > t2[0][0]:
```

```
8           return [t1[0]] + g(t1[1:], t2)
9       else:
10          return [t2[0]] + g(t1, t2[1:])
11
12  def h(t):
13      n = len(t)
14      if len(t) < 2:
15          return t
16      else:
17          t1, t2 =  t[:n//2], t[n//2:]
18          return g(h(t1), h(t2))
```

A5. De quel tri s'agit-il ? Quel ordre résulte de ce tri ?

Solution : Il s'agit du tri fusion. La ligne 7 du code indique que l'ordre sera décroissant en fonction du nombre de fonctionnalité de `C`, c'est-à-dire la valeur du premier élément de chaque tuple.

A6. Quelle est la complexité de cet algorithme dans le pire et le meilleur des cas ? Pour justifier, on s'appuiera sur la récurrence que vérifie la complexité sans développer de calculs.

Solution : Sa complexité est toujours en $O(n \log n)$ quelque soit le cas, si la fonction fusion est linéaire. La justification est la forme de la récurrence que vérifie la complexité : $T(n) = n+2T(n/2)$ (cf. cours) typique des algorithmes diviser pour régner avec une division par deux et deux appels récursifs.

A7. Choisir une des deux stratégies et coder une fonction de prototype `puces(C, smax)` qui l'implémente. Préciser la stratégie choisie et utiliser la fonction de tri précédente.

Solution :

```
1   def greedy_surface(circuits, s_max):
2       s_tot = 0
3       f_tot = 0
4       selection = []
5       circuits = h(circuits)
6       i = 0
7       while i < len(circuits) and s_tot <= s_max:
8           f, s = circuits[i]
9           if s_tot + s <= s_max:
10              selection.append((f, s))
11              s_tot += s
12              f_tot += f
13          i += 1
14      return selection, f_tot, s_tot
15
16
17  def greedy_ratios(circuits, s_max):
18      s_tot = 0
```

```
    f_tot = 0
    selection = []
    ratios = h([(f / s, f, s) for f, s in circuits])
    i = 0
    while i < len(ratios) and s_tot <= s_max:
        r, v, p = ratios[i]
        if s_tot + p <= s_max:
            selection.append((v, p))
            s_tot += p
            f_tot += v
        i += 1
    return selection, f_tot, s_tot
```

B Jump game

Le jeu du saut est un jeu à un seul joueur.

- On dispose d'une liste de n entiers naturels numérotée de 0 à $n-1$.
- La position de départ est la case d'indice 0.
- La position d'arrivée est la case d'indice $n-1$.

Le but du jeu est d'atteindre la position d'arrivée depuis la position de départ **en effectuant un minimum de saut** et en respectant la règle suivante :
chaque élément de la liste représente la longueur maximale du saut qu'il est possible d'effectuer.

■ **Exemple 1 — Exemple de partie de Jump Game.** On dispose du jeu suivant :

0	1	2	3	4	5	6	7	8	9	10
1	2	6	8	6	3	6	0	1	2	9

Une partie possible est la suivante :

1. La partie commence sur la case d'indice 0.
2. Au premier coup, le joueur ne peut avancer que sur la case d'indice 1.
3. Au second coup, le joueur peut atteindre les cases d'indice 2 ou 3. On ne sait pas pourquoi, mais il choisit la case d'indice 2.
4. Au troisième coup, le joueur choisit d'aller sur la case d'indice 8.
5. Au quatrième coup, il ne peut aller qu'en 9.
6. Au dernier coup, il choisit d'aller en 10 et la partie est finie.

On peut résumer sa partie par la liste de positions `[1,2,8,9,10]`. Naturellement, il aurait pu gagner en moins de coups en choisissant `[1, 3, 10]`.
Ce jeu est représenté par la suite par une liste Python : `[1, 2, 6, 8, 6, 3, 6, 0, 1, 2, 9]`.

B8. Écrire une fonction de signature `case_max(jeu : list[int], c : int)—> int` qui renvoie l'indice de la case la plus éloignée que le joueur peut atteindre à partir de la case d'indice `c` dans le jeu `jeu`. Si jamais le joueur ne peut plus jouer, c'est-à-dire s'il est tombé sur une case nulle, alors la fonction renvoie `None`.

Solution :

```
def case_max(jeu, c):
    n = len(jeu)
    saut_max = jeu[c]
    if saut_max == 0:
        return None
    if c + saut_max < n:
        return c + saut_max
    else:
        return n - 1
```

On définit une stratégie $\mathcal{S}$ comme suit :

> « à partir d'une case d'indice `c`, le joueur décide d'aller systématiquement sur la case plus proche de la position d'arrivée, c'est-à-dire qu'il décide d'effectuer le plus grand saut possible. »

B9. Comment peut-on qualifier la stratégie $\mathcal{S}$? Pourquoi ?

Solution : C'est une stratégie gloutonne car elle choisit de se rapprocher de la position d'arrivée en choisissant de réduire la distance le plus vite possible localement. Elle choisit l'optimal local, le saut de longueur maximale à chaque fois sans analyser les autres possibilités, même si ce n'est pas la meilleure stratégie globalement.

B10. La stratégie $\mathcal{S}$ est-elle toujours optimale ? Justifier votre réponse en donnant un contre-exemple ou en prouvant l'optimalité.

Solution : Sur le jeu `[2,3,1,1,4]`, la stratégie $\mathcal{S}$ n'aboutit pas à la solution optimale puisqu'elle effectue trois sauts alors qu'on peut gagner en deux sauts. Elle n'est donc pas toujours optimale.

B11. Écrire une fonction de signature `jouer_S(jeu)-> list[int]` qui renvoie la liste des positions empruntées par un joueur qui suit la stratégie $\mathcal{S}$. Si la position du joueur n'est pas l'arrivée et si elle ne permet plus d'avancer, la fonction renvoie `None`. On garantira par une assertion que le jeu possède au moins une case de départ. Par exemple, pour le jeu `[1, 2, 6, 8, 6, 3, 6, 0, 1, 2, 9]`, cette fonction renvoie `[1, 3, 10]`.

Solution :

```
def jouer_S(jeu):
    assert len(jeu) > 0
    n = len(jeu)
    solution = []
    c = 0
    # il faut que  c < n et c'est fini lorsque  c == n - 1:
    while c < n - 1:
        c = case_max(jeu, c)
        if c != None:
            solution.append(c)
```

```
        else:
            return None
    return solution
```

On décline une autre stratégie $\mathcal{D}$ comme suit :

> « à partir d'une case d'indice `c`, le joueur décide d'effectuer le meilleur saut, c'est-à-dire celui qui l'amène sur la case dont l'entier est le plus grand. En cas d'égalité des entiers, il choisit la case la plus proche de la position d'arrivée.»

B12. Comment peut-on qualifier la stratégie $\mathcal{D}$?

Solution : $\mathcal{D}$ est également une stratégie gloutonne car elle n'examine pas toutes les possibilités et effectue systématiquement un choix optimal localement.

B13. La stratégie $\mathcal{D}$ est-elle toujours optimale? Justifier votre réponse en donnant un contre-exemple ou en prouvant l'optimalité.

Solution : Sur le jeu `[2,2,1,1,2,3,1]`, la stratégie $\mathcal{D}$ n'aboutit pas à la solution optimale puisqu'elle effectue cinq sauts alors que la stratégie $\mathcal{S}$ permet de gagner en quatre sauts. Elle n'est donc pas toujours optimale.

B14. Écrire une fonction de signature `prochaine_case(jeu, c)->int` qui renvoie l'indice de la prochaine case calculée selon la stratégie $\mathcal{D}$. Si jamais le joueur ne peut plus jouer, cette fonction renvoie `None`.

Solution :

```
def prochaine_case(jeu, c):
    plus_loin = case_max(jeu, c)
    if plus_loin != None: # on peut sauter
        meilleur_saut = 0
        prochaine = c
        for j in range(c + 1, plus_loin + 1):
            if jeu[j] >= meilleur_saut:
                meilleur_saut = jeu[j]
                prochaine = j
        return prochaine
    else:
        return None
```

B15. Que faudrait-il modifier dans le code de `jouer_S` pour créer une fonction `jouer_D` qui renvoie la liste des positions empruntées par un joueur qui suit la stratégie $\mathcal{D}$?

Solution : Pas grand chose : remplacer `case_max` par `prochaine_case`.

```
def jouer_D(jeu):
    assert len(jeu) > 0
```

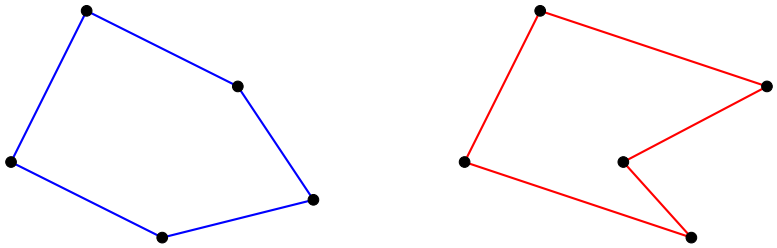

FIGURE 1 – L'ensemble de points reliés en bleu forment une partie convexe du plan. L'ensemble des points reliés en rouge une partie non convexe

```
    n = len(jeu)
    solution = []
    c = 0
    while c < n - 1:
        c = prochaine_case(jeu, c)
        if c != None:
            solution.append(c)
        else:
            return None
    return solution
```

C Déterminer une enveloppe complexe

Les concepts de partie convexe du plan et d'enveloppe convexe sont illustrés sur les figures 1 et 2 et définis ci-dessous.

■ **Définition 1 — Partie convexe du plan.** Une partie C de $\mathbb{R}^2$ est dite convexe si, pour tout couple de points (a, b) de C, le segment $[a, b]$ est entièrement contenu dans C.

■ **Définition 2 — Enveloppe convexe.** Soit A une partie du plan. L'enveloppe convexe de A est l'intersection de toutes les parties convexes de E qui contiennent A. C'est aussi la plus petite partie convexe de E contenant A.

Les applications du concept d'enveloppe convexe sont multiples :

- planifier une trajectoire en robotique,
- délimiter une partie du plan d'une manière optimale,
- reconnaitre une forme sur une image,
- mailler une structure mécanique…

On dispose d'un ensemble de points `points` du plan sous la forme d'une liste de tuples : le premier élément du tuple est l'abscisse du point, le second l'ordonnée. L'objectif de cette section est de coder plusieurs algorithmes permettant de construire l'enveloppe convexe de l'ensemble de points.

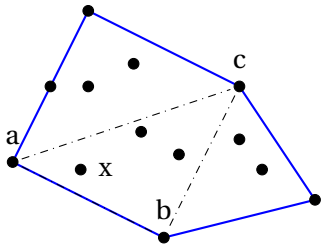

FIGURE 2 – Enveloppe convexe (en bleu) d'un ensemble de points.

a Algorithme naïf

Dans un premier temps, on procède d'une manière naïve : on teste chaque point de l'ensemble pour savoir s'il est contenu dans un triangle de points de l'ensemble. Si c'est le cas, alors ce point est à l'intérieur de l'enveloppe (cf. point x sur la figure 2), sinon il se trouve sur l'enveloppe (cf. points a, b ou c sur la figure 2).

On peut savoir si un point c se trouve à gauche ou à droite d'une droite $(\overrightarrow{ab})$ en calculant :

$$d = (x_b - x_a) \times (y_c - y_a) - (y_b - y_a) \times (x_c - x_a) \tag{1}$$

Si $d = 0$ le point c est sur la droite $(\overrightarrow{ab})$

Si $d < 0$ le point c est à droite de $(\overrightarrow{ab})$

Si $d > 0$ le point c est à gauche de $(\overrightarrow{ab})$

R Sur la figure 2, le point x se trouve à gauche de $(\overrightarrow{ab})$ mais à droite de $(\overrightarrow{ba})$.

C16. Écrire une fonction de signature `orientation(a :(float,float), b :(float,float), c :(float,float) )-> float` qui renvoie la grandeur d de l'équation 1. On cherche à savoir si le point `c` est à gauche ou à droite de $(\overrightarrow{ab})$.

Solution :

```
def orientation(a, b, c):
    """
    Renvoie:
        - > 0 c à gauche
        - < 0 c à droite
        - 0 c sur la droite (colinéaires)
    """
    return (b[0] - a[0]) * (c[1] - a[1]) - (b[1] - a[1]) * (c[0] - a[0])
```

C17. Écrire une fonction de signature `interieur(a, b, c, x)-> bool` qui statue sur le fait que le point x est à l'intérieur du triangle (a, b, c). Un point sur un segment du triangle est considéré à l'extérieur du triangle. On garantira par une assertion que les points a, b, c sont tous différents.

Solution :

```
def interieur(a, b, c, x):
    assert a != b and b != c and c != a
    s1 = orientation(a, x, b)
    s2 = orientation(b, x, c)
    s3 = orientation(c, x, a)
    c1 = s1 > 0 and s2 > 0 and s3 > 0
    c2 = s1 < 0 and s2 < 0 and s3 < 0
    return c1 or c2
```

C18. Écrire une fonction de signature `ec_naif(points : list[(float, float)])-> list[(float, float)]` où `points` est un ensemble de points du plan et qui renvoie l'enveloppe convexe de cet ensemble sous la forme d'une liste de points. Cette fonction applique l'algorithme naïf décrit précédemment. Par exemple, pour la liste de points `[(0,0), (0.5,1.5), (2,2), (0,2), (2,0)]`, la fonction renvoie `[(0,0), (2,2), (0,2), (2,0)]`.

Solution :

```
def ec_naif(points):
    n = len(points)
    ec = [True for _ in range(n)]
    for i in range(n):
        for j in range(i + 1, n):
            for k in range(j + 1, n):
                for x in range(n):
                    if interieur(points[i], points[j], points[k], points[x]):
                        ec[x] = False
    enveloppe = []
    for i in range(n):
        if ec[i]:
            enveloppe.append(points[i])
    return enveloppe
```

R Le test `intérieur` échoue lorsque le point testé est sur un côté du triangle. Dans la pratique, ce cas est rare, car les points sont flottants. Mais il existe. Plutôt que d'améliorer cette approche naïve, on préfère développer des algorithmes plus performants.

C19. Quelle est la complexité de cette approche naïve ?

Solution : La fonction `interieur` s'effectue en un temps constant c. Donc, il suffit de calculer le nombre d'itérations des quatre boucles imbriquées.

$$C(n) = \sum_{i=0}^{n} \sum_{j=i+1}^{n} \sum_{k=j+1}^{n} \sum_{x=0}^{n} c \tag{2}$$

$$= \sum_{x=0}^{n} \sum_{i=0}^{n} \sum_{j=i+1}^{n} \sum_{k=j+1}^{n} c \tag{3}$$

$$= c \sum_{x=0}^{n} \sum_{i=0}^{n} \sum_{j=i+1}^{n} n - j \tag{4}$$

$$= c \sum_{x=0}^{n} \sum_{i=0}^{n} \sum_{k=0}^{n-i-1} k \tag{5}$$

$$= c \sum_{x=0}^{n} \sum_{i=0}^{n} \frac{1}{2}(n-i-1)(n-i) \tag{6}$$

$$= c \sum_{x=0}^{n} \sum_{j=0}^{n} \frac{1}{2}(j-1)j \tag{7}$$

$$= c \sum_{x=0}^{n} O(n^3) = O(n^4) \tag{8}$$

$$\tag{9}$$

b Papier cadeau

L'algorithme de Jarvis [1] est une autre méthode pour obtenir l'enveloppe convexe. L'idée est d'envelopper l'ensemble des points comme on le ferait pour emballer un papier cadeau : en deux dimensions, on accroche l'extrémité d'un fil sur le point d'abscisse minimum puis on le tend et on tourne dans le sens horaire autour du nuage de points jusqu'à revenir au point de départ.

Cet algorithme repose sur deux constatations :

1. le point le plus à gauche (abscisse minimum) fait nécessairement partie de l'enveloppe,
2. si un point p fait partie de l'enveloppe, alors le point suivant est le point q tel que $(\overrightarrow{pq})$ est la droite la plus inclinée vers la gauche (dans le sens anti-horaire).

C20. Écrire une fonction de signature `point_en_haut_a_gauche(points)—> int` qui renvoie l'indice du points d'abscisse minimum dans la liste de points. En cas d'égalité, on choisit le point le plus haut, c'est-à-dire d'ordonnée la plus grande.

Solution :

```
1 def point_en_haut_a_gauche(points):
2     if points == []:
3         return None
4     else:
5         i_min = 0
6         m = points[0][0]
7         for k in range(len(points)):
```

1. marche de Jarvis ou algorithme du papier cadeau

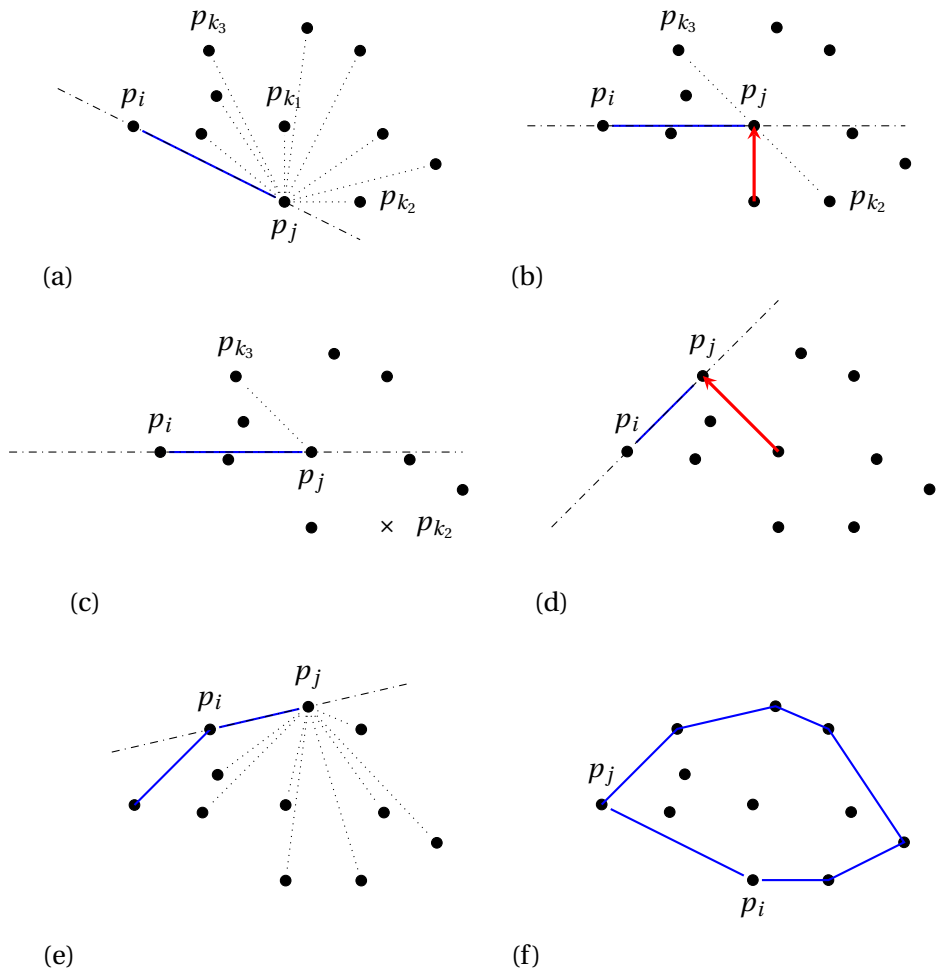

FIGURE 3 – Début de la marche de Jarvis : étape (a), (b), (c) et (d) pour $k = k_1, k_2, k_3 \ldots$ Le points p_{k_2} n'est pas à gauche de $(\overrightarrow{p_i, p_j})$, par contre c'est le cas pour p_{k_1} et p_{k_3}, le plus à gauche. L'étape (f) marque la fin de l'algorithme : tous les points de l'enveloppe ont été trouvés.

```
8               if points[k][0] < m:
9                   m = points[k][0]
10                  i_min = k
11              elif points[k][0] == m:
12                  if points[k][1] > points[i_min][1]:
13                      i_min = k
14          return i_min
```

C21. Écrire une fonction de signature `prochain_point(points, i)-> int` qui renvoie l'indice du prochain point de l'enveloppe convexe. On suppose que `i` est l'indice du dernier point trouvé de l'enveloppe. On procèdera de la manière suivante : si n est la taille de la liste `points`, on part des points d'indice i et `j = (i+1)%n`. Puis, pour chaque points d'indice k, si le point k est un point à gauche de $(\overrightarrow{\text{points}[i]\text{points}[j]})$, alors j devient k et on continue avec les points k restant. On remonte ainsi de

proche en proche vers le prochain point de l'enveloppe convexe. La figure 3 illustre cet algorithme.

Solution :

```
def prochain_point(points, i):
    n = len(points)
    j = (i + 1) % n
    for k in range(n):
        if orientation(points[i], points[k], points[j]) > 0:
            j = k
    return j
```

C22. Écrire une fonction de signature `ec_jarvis(points)` qui renvoie l'enveloppe convexe de la liste de points. On procède de proche en proche en partant du point le plus à gauche et en cherchant le point suivant de l'enveloppe. Lorsque le point suivant est le point de départ, l'exploration est finie et on a trouvé l'enveloppe convexe.

Solution :

```
def ec_jarvis(points):
    assert len(points) > 2
    n = len(points)
    ec = []
    most_left = point_en_haut_a_gauche(points)
    ec.append(points[most_left])
    j = prochain_point(points, most_left)
    while j != most_left:
        ec.append(points[j])
        # le point suivant le plus sur la gauche
        j = prochain_point(points, j)
    return ec
```

C23. Quelle est la complexité de l'algorithme de Jarvis ? On pourra noter n le nombre de points de la liste `points` et h le nombre de points de l'enveloppe convexe.

Solution : Soit n le nombre de points de la liste `points`. La fonction `point_en_haut_a_gauche` est en $O(n)$ car elle parcourt tous les points de la liste. La fonction `prochain_point` est en $O(n)$ car elle parcourt tous les points de la liste. La boucle `while` de la fonction `ec_jarvis` est répétée autant de fois qu'il y a de points sur l'enveloppe convexe. Soit h ce nombre de points. La complexité de `ec_jarvis` est donc en $O(n+nh) = O(nh)$.

R On peut le faire en $O(n \log h)$...