Chapitre 9

Algorithmique

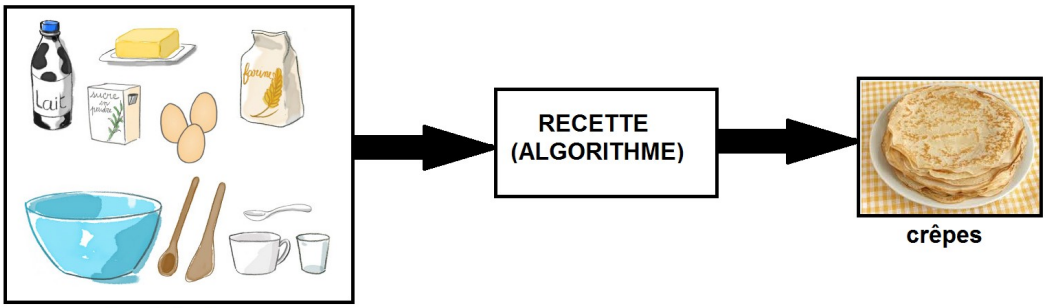
On désigne par algorithmique l'ensemble des activités logiques qui relèvent des algorithmes ; en particulier, en informatique, cette discipline désigne l'ensemble des règles et des techniques qui sont impliquées dans la définition et la conception des algorithmes.

9.1. Quelques définitions

Al Khwarizmi
(780 ? - 850?)

Le mot « algorithme » vient du nom du mathématicien **Al Khwarizmi**, qui, au 9ème siècle écrivit le premier ouvrage systématique sur la solution des équations linéaires et quadratiques. La notion d'algorithme est donc historiquement liée aux manipulations numériques, mais elle s'est progressivement développée pour porter sur des objets de plus en plus complexes : des textes, des images, des formules logiques, des objets physiques, etc.

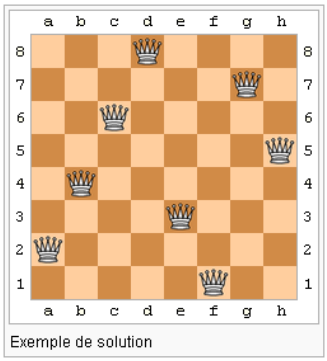
Un algorithme est un énoncé d'une suite d'opérations permettant de donner la réponse à un problème.



ingrédients

- Si les opérations s'exécutent sur plusieurs processeurs en parallèle, on parle d'algorithme parallèle.
- Si les tâches s'exécutent sur un réseau de processeurs on parle d'algorithme distribué.
- Un algorithme qui contient un appel à lui-même est dit récursif.
- Un algorithme glouton est un algorithme qui suit le principe de faire, étape par étape, un choix optimum local, dans l'espoir d'obtenir un résultat optimum global. Dans les cas où l'algorithme ne fournit pas systématiquement la solution optimale, il est appelé une heuristique gloutonne.
- En optimisation combinatoire, théorie des graphes et théorie de la complexité, une heuristique est un algorithme qui fournit rapidement (en temps polynomial) une solution réalisable, mais pas nécessairement optimale, pour un problème d'optimisation difficile. Une heuristique, ou méthode approximative, est donc le contraire d'un algorithme exact qui trouve une solution optimale pour un problème donné. L'usage d'une heuristique est pertinente pour calculer une solution approchée d'un problème et ainsi accélérer le processus de résolution exacte.

9.2. Le problème des huit dames

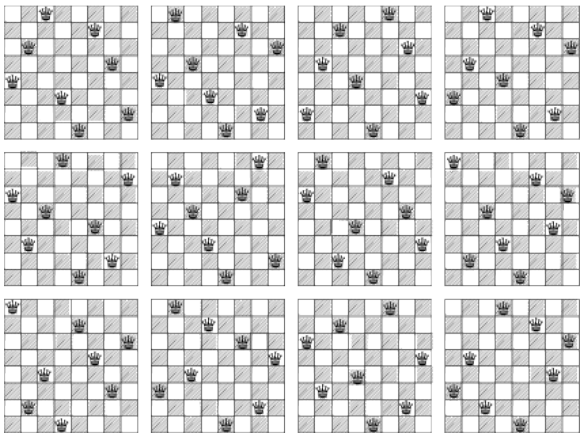


Exemple de solution

Le but du problème des huit dames, est de placer huit dames d'un jeu d'échecs sur un échiquier de 8 x 8 cases sans que les dames ne puissent se menacer mutuellement, conformément aux règles du jeu d'échecs. Par conséquent, deux dames ne devraient jamais partager la même rangée, colonne, ou diagonale (voir dessin ci-contre).

Durant des années, beaucoup de mathématiciens, y compris **Gauss** ont travaillé sur ce problème, qui est un cas particulier du problème généralisé des *n*-dames, posé en 1850 par Franz **Nauck**, et qui est de placer *n* dames « libres » sur un échiquier de *n* x *n* cases. En 1874, S. **Gunther** proposa une méthode pour trouver des solutions en employant des déterminants, et J. W. L. **Glaisher** affina cette approche.

Le problème des huit dames a 92 solutions distinctes, ou seulement 12 solutions si l'on tient compte de transformations telles que des rotations ou des réflexions.



Le problème des huit dames est un bon exemple de problème simple mais non évident. Pour cette raison, il est souvent employé comme support de mise en œuvre de différentes techniques de programmation, y compris d'approches non traditionnelles de la programmation telles que la programmation par contraintes, la programmation logique ou les algorithmes génétiques.

9.2.1. Algorithme naïf

L'algorithme naïf de recherche exhaustive teste toutes les manières possibles de placer une dame par colonne, pour retirer toutes celles où des dames se menacent mutuellement.

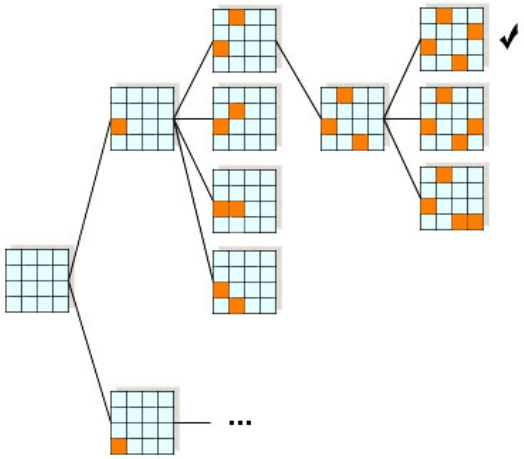
Il y a 8^8 = 16'777'216 placements à explorer.

Exercice 9.1

Écrivez en pseudo-code, puis programmez l'algorithme naïf pour trouver les 92 solutions au problème des huit dames. Vous représenterez la position par une liste de nombres de 0 à 7. Ces nombres indiquent la ligne sur laquelle la dame se trouve pour la colonne correspondante. Par exemple, la 12ème solution du graphique ci-dessus sera représentée par la liste : [2, 4, 1, 7, 0, 6, 3, 5].

9.2.2. Recherche en profondeur

La recherche en profondeur consiste à trouver les solutions en plaçant les dames de gauche à droite. Prenons pour simplifier un exemple sur un damier 4 x 4. Imaginons que toutes les solutions avec la première dame sur la première ligne ont été trouvées (voir dessin ci-dessous).



On veut maintenant placer la première dame sur la 2ème ligne.

Plaçons la deuxième dame sur la ligne 1. Il y a conflit : STOP. On ne cherche pas plus loin. Idem pour les lignes 2 et 3. La seule solution possible est la 4ème ligne.

Plaçons maintenant la troisième dame. On peut la mettre sur la première ligne sans conflit.

Plaçons alors la 4ème et dernière dame. La seule ligne possible est la 3ème. On a une solution : [1,3,0,2].

On remonte alors dans l'arbre : la troisième dame ne peut se placer sur aucune autre ligne.

Remontons encore d'un cran.

On a déjà essayé toutes les lignes pour la deuxième dame.

Remontons encore d'un cran : plaçons la première dame sur la 3ème ligne et recommençons une recherche en profondeur...

Exercice 9.2

Programmez la recherche en profondeur expliquée ci-dessus. Comptez le nombre de situations (intermédiaires et finales) analysées.

9.2.3. Méthode heuristique

Un algorithme de « réparation itérative » commence typiquement à partir d'un placement de toutes les dames sur l'échiquier, par exemple avec une seule dame par ligne et par colonne. Il essaie ensuite toutes les permutations des colonnes (il n'y a que des conflits diagonaux à tester) pour ne garder que les configurations sans conflits.

Il y a 8! = 40'320 placements à explorer.

Exercice 9.3

Placez initialement les 8 dames sur la diagonale de l'échiquier, puis échangez deux colonnes choisies au hasard. Répétez l'opération jusqu'à ce qu'**une** solution soit trouvée.

Exercice 9.4

Modifiez le programme de l'exercice 9.3 pour trouver les 92 solutions.

Le programme ci-dessous pourra vous aider. Après avoir vu ce qu'il produit, insérez-le dans votre programme.

```
from itertools import permutations

for p in permutations([0,1,2]):
    print(p)
```

9.3. Algorithmes gloutons

Un algorithme glouton (en anglais : *greedy*) est un algorithme qui suit le principe de faire, étape par étape, un choix optimum local, dans l'espoir d'obtenir un résultat optimum global. Il n'y a pas de retour en arrière : à chaque étape de décision dans l'algorithme, le choix qui semble le meilleur à ce moment est effectué et est définitif. Les algorithmes gloutons servent surtout à résoudre des problèmes d'optimisation.

Exemple : rendu de monnaie

Dans le problème du rendu de monnaie (donner une somme avec le moins possible de pièces), l'algorithme consistant à répéter le choix de la pièce de plus grande valeur qui ne dépasse pas la somme restante est un algorithme glouton.

Suivant le système de pièces, l'algorithme glouton est optimal ou pas. Dans le système de pièces européen (en centimes : 1, 2, 5, 10, 20, 50, 100, 200), où l'algorithme glouton donne la somme suivante pour 37 : 20+10+5+2, on peut montrer que l'algorithme glouton donne toujours une solution optimale.

Dans le système de pièces (1, 3, 4), l'algorithme glouton n'est pas optimal. En effet, il donne pour 6 : 4+1+1, alors que c'est 3+3 qui est optimal.

Exercice 9.5

Une route comporte n stations-service, numérotées dans l'ordre du parcours, de 0 à n–1. La première est à une distance d [0] du départ, la deuxième est à une distance d [1] de la première, la troisième à une distance d [2] de la deuxième, etc. La fin de la route est à une distance d [n] de la n-ième et dernière station-service.

Un automobiliste prend le départ de la route avec une voiture dont le réservoir d'essence est plein. Sa voiture est capable de parcourir une distance r avec un plein.

Question 9.5.1

Donnez une condition nécessaire et suffisante pour que l'automobiliste puisse effectuer le parcours. On la supposera réalisée par la suite.

Question 9.5.2

Prenez 17 stations-service avec les distances d = [23, 40, 12, 44, 21, 9, 67, 32, 51, 30, 11, 55, 24, 64, 32, 57, 12, 80] et r = 100.

L'automobiliste désire faire le plein le moins souvent possible. Écrivez en pseudo-code, puis programmez une fonction Python `rapide` qui détermine à quelles stations-service il doit s'arrêter.

Exercice 9.6

Un cambrioleur entre par effraction dans une maison. Il n'est capable de porter que K kilos : il lui faudra donc choisir entre les différents objets de valeur, afin d'amasser le plus gros magot possible.

On supposera dans un premier temps que les objets sont fractionnables (on peut en prendre n'importe quelle quantité, c'est le cas d'un liquide ou d'une poudre). Il y a n matières différentes, numérotées de 0 à n–1, la i-ème ayant un prix $p[i]$ par kilo. La quantité disponible de cette matière est $q[i]$. On suppose que tous les prix sont différents deux à deux.

Question 9.6.1

Écrivez en pseudo-code un algorithme qui donne un choix optimal pour le voleur. Ce choix est-il unique ?

Programmez une fonction `voleur` en Python qui reprenne cet algorithme (vous pourrez supposer que le tableau p est trié).

Prenez par exemple :
```
p = [43, 40, 37, 33, 28, 25, 20, 17, 14, 13]
q = [ 7,  6, 12, 11,  2, 23,  1,  4, 24, 43]
K = 55
```

Question 9.6.2

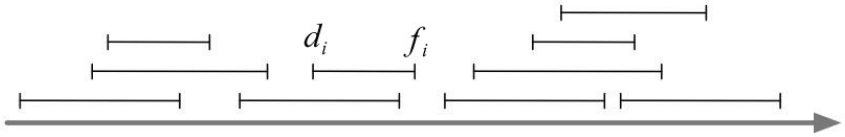
On suppose maintenant que les objets sont non fractionnables (c'est le cas d'un vase ou d'un téléviseur). Le i-ème objet vaut un prix $p[i]$ et pèse un poids $q[i]$.

Proposez une méthode dérivée de la question 1 (sans la programmer).

Donne-t-elle un choix optimal ?

Exercice 9.7

Dans un cinéma, chaque séance i est caractérisée par l'intervalle (d_i , f_i), où d_i est l'heure de début et f_i l'heure de fin. On peut représenter ces séances sur un schéma d'intervalles :



Exemple de planning des séances de cinéma

Vous voulez assister au maximum de séances dans une journée.

Vous considérez trois critères pour classer les séances, de la plus petite valeur à la plus grande :

1. critère A : l'heure de début de la séance (d_i)
2. critère B : la durée de la séance (f_i - d_i)
3. critère C : l'heure de fin de la séance (f_i)

a. Décrivez en pseudo-code un algorithme glouton permettant de choisir les séances, après les avoir classées selon l'un des critères ci-dessus.

b. Pour chacun des trois critères de classement, exhibez un cas (s'il en existe un) où votre algorithme ne donnera pas un choix optimal. Donnez un exemple avec 5 séances sous forme d'un schéma d'intervalles comme celui de l'énoncé du problème.

c. Appliquez votre algorithme aux séances ci-dessous. Donner le résultat pour chacun des trois critères de classement.

Séance (i)	1	2	3	4	5	6	7	8	9	10
Début (d_i)	9h	9h15	10h	13h	15h	15h15	16h	17h30	18h	19h30
Fin (f_i)	11h	10h50	11h20	15h25	16h40	18h15	18h05	19h	20h10	22h
Durée (f_i - d_i)	120	95	80	145	100	180	125	90	130	150

d. Vous voulez absolument assister à la séance 9. Modifiez votre algorithme pour intégrer cette contrainte supplémentaire, puis répondez à nouveau à la question **c.**

http://ow.ly/4tpWm

9.4. Algorithmes de tri

Une des opérations les plus courantes en programmation est le tri d'objets. Les ordres les plus utilisés sont l'ordre numérique et l'ordre lexicographique.

Les principales caractéristiques qui permettent de différencier les algorithmes de tri sont la complexité algorithmique (voir § 8.5) et les ressources nécessaires (notamment en terme d'espace mémoire utilisé).

On peut montrer que la complexité temporelle en moyenne et dans le pire des cas d'un algorithme basé sur une fonction de comparaison ne peut pas être meilleure que $O(n\log(n))$. Les tris qui ne demandent que $O(n\log(n))$ comparaisons en moyenne sont alors dits optimaux.

Dans les algorithmes ci-dessous, on effectuera des tris par ordre croissant.

9.4.1. Tri par sélection

Le tri par sélection est un des algorithmes de tri les plus triviaux.

On recherche le plus grand élément que l'on va replacer à sa position finale, c'est-à-dire en dernière position.

Puis on recherche le second plus grand élément que l'on va placer en avant-dernière position, etc., jusqu'à ce que le tableau soit entièrement trié.

```
def swap(l,i,j):
    # echange 2 valeurs d'une liste
    l[i],l[i] = l[j], l[i]

def tri_selection(l):
    for i in range(len(l)-1):
        mini=i
        for j in range(i+1,len(l)):
            if l[j]<l[mini]: mini=j
        swap(l,i,mini)
```

Complexité

- Meilleur des cas : $O(n^2)$ quand le tableau est déjà trié.
- Pire cas : $O(n^2)$ quand le tableau est trié en ordre inverse.
- En moyenne : $O(n^2)$.

9.4.2. Tri à bulles (Bubble sort)

Le tri à bulles est un algorithme de tri qui consiste à faire remonter progressivement les plus petits éléments d'une liste, comme les bulles d'air remontent à la surface d'un liquide.

L'algorithme parcourt la liste, et compare les couples d'éléments successifs. Lorsque deux éléments successifs ne sont pas dans l'ordre croissant, ils sont échangés. Après chaque parcours complet de la liste, l'algorithme recommence l'opération. Lorsqu'aucun échange n'a lieu pendant un parcours, cela signifie que la liste est triée : l'algorithme peut s'arrêter.

Cet algorithme est souvent enseigné en tant qu'exemple algorithmique. Cependant, il présente une complexité en $O(n^2)$ dans le pire des cas (où n est la longueur de la liste), ce qui le classe parmi les mauvais algorithmes de tri. Il n'est donc quasiment pas utilisé en pratique.

```
def swap(l,i,j):
    # echange 2 valeurs d'une liste
    l[i],l[j] = l[j],l[i]

def tri_a_bulles(l):
    for i in range(len(l)):
        for j in reversed(range(i,len(l))):
            if l[j]<l[j-1]:
                swap(l,j-1,j)
```

Complexité

- Meilleur cas : $O(n)$ quand le tableau est trié.
- Pire des cas : $O(n^2)$ quand le tableau est trié en ordre inverse $(n-1)\cdot(n-1) = O(n^2)$

9.4.3. Tri par insertion

Le tri par insertion est le tri le plus efficace sur des listes de petite taille. C'est pourquoi il est utilisé par d'autres méthodes comme le *Quicksort* (voir § 9.4.4). Il est d'autant plus rapide que les données sont déjà triées en partie dans le bon ordre.

Le principe de ce tri est très simple : c'est le tri que toute personne utilise naturellement quand elle a des dossiers (ou n'importe quoi d'autre) à classer. On prend un dossier et on le met à sa place parmi les dossiers déjà triés. Puis on recommence avec le dossier suivant.

Pour procéder à un tri par insertion, il suffit de parcourir une liste : on prend les éléments dans l'ordre. Ensuite, on les compare avec les éléments précédents jusqu'à trouver la place de l'élément qu'on considère. Il ne reste plus qu'à décaler les éléments du tableau pour insérer l'élément considéré à sa place dans la partie déjà triée.

```
def tri_insertion(liste):
    j, n = 1, len(liste)
    while j != n:
        i = j - 1
        temp = liste[j]
        while i > -1 and liste[i] > temp:
            liste[i+1] = liste[i]
            i = i - 1
        liste[i+1] = temp
        j = j + 1
```

Complexité

- Pire cas : $O(n^2)$ quand le tableau est trié en ordre inverse
- Moyenne : $O(n^2)$

9.4.4. Quicksort

Charles Antony Richard **Hoare** (né en 1934)

Le Quicksort est une méthode de tri inventée par Sir Charles Antony Richard **Hoare** en 1961 et fondée sur la méthode de conception « diviser pour régner ». Il peut être implémenté sur un tableau ou sur des listes ; son utilisation la plus répandue concerne tout de même les tableaux.

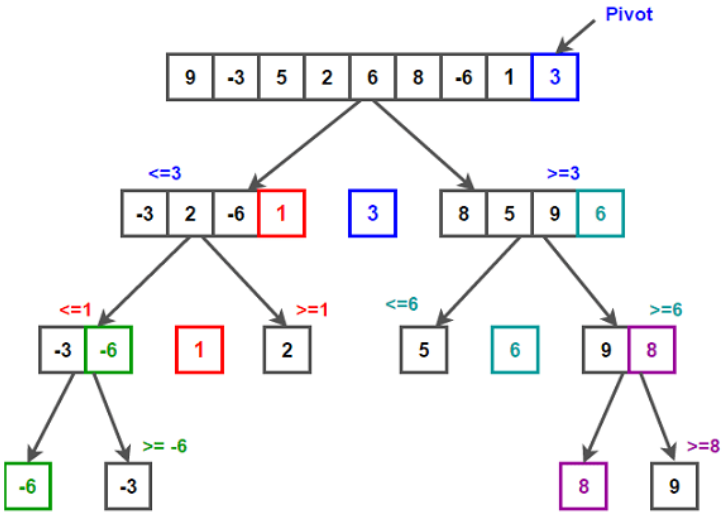
Le Quicksort est un tri dont la complexité moyenne est en $O(n\log(n))$, mais dont la complexité dans le pire des cas est un comportement quadratique en $O(n^2)$. Malgré ce désavantage théorique, c'est en pratique un des tris les plus rapide pour des données réparties aléatoirement. Les entrées donnant lieu au comportement quadratique dépendent de l'implémentation de l'algorithme, mais sont souvent (si l'implémentation est maladroite) les entrées déjà presque triées. Il sera plus avantageux alors d'utiliser le tri par insertion.

La méthode consiste à placer un élément du tableau (appelé pivot) à sa place définitive, en permutant tous les éléments de telle sorte que tous ceux qui lui sont inférieurs soient à sa gauche et que tous ceux qui lui sont supérieurs soient à sa droite. Cette opération s'appelle le partitionnement.

Pour chacun des sous-tableaux, on définit un nouveau pivot et on répète l'opération de partitionnement. Ce processus est répété récursivement, jusqu'à ce que l'ensemble des éléments soit trié.

```
def partition(tab, debut, fin):
    while debut < fin:
        while debut < fin:
            if tab[debut] > tab[fin]:
                tab[debut], tab[fin] = tab[fin], tab[debut]
                break
            fin = fin - 1
        while debut < fin:
            if tab[debut] > tab[fin]:
                tab[debut], tab[fin] = tab[fin], tab[debut]
                break
            debut = debut + 1
    return debut

def quicksort(tab, debut=None, fin=None):
    if debut is None: debut = 0
    if fin is None: fin = len(tab)
    if debut < fin:
        i = partition(tab, debut, fin-1)
        quicksort(tab, debut, i)
        quicksort(tab, i+1, fin)
```



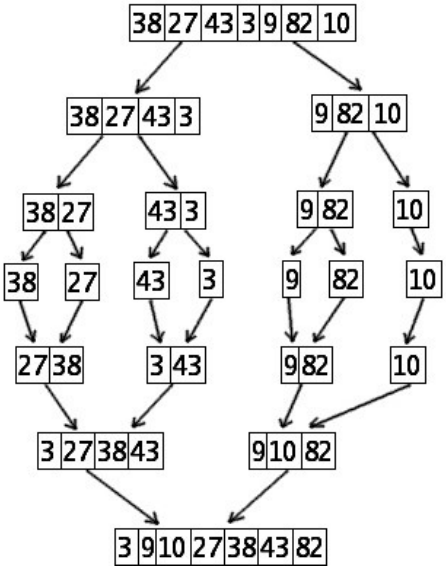
Dans la pratique, pour les partitions avec un faible nombre d'éléments (jusqu'à environ 15 éléments), on a souvent recours à un tri par insertion qui se révèle plus efficace que le Quicksort.

Le problème le plus important est le choix du pivot. Une implémentation du Quicksort qui ne choisit pas adéquatement le pivot sera très inefficace pour certaines entrées. Par exemple, si le pivot est toujours le plus petit élément de la liste, Quicksort sera aussi inefficace qu'un tri par sélection.

Complexité

- Pire des cas : $O(n^2)$ quand le tableau est trié en ordre inverse
- En moyenne et dans le meilleur des cas : $O(n\log(n))$

9.4.5. Tri par fusion (Mergesort)



Le tri par fusion repose sur le fait que, pour fusionner deux listes/tableaux trié(e)s dont la somme des longueurs est n, $n-1$ comparaisons au maximum sont nécessaires. Pour aller aussi vite que le *Quicksort*, il a donc besoin d'utiliser $O(n)$ mémoire supplémentaire, mais il a l'avantage d'être stable c'est-à-dire de ne pas mélanger ce qui est déjà trié.

L'algorithme peut être décrit récursivement :

1. On découpe en deux parties à peu près égales les données à trier.
2. On trie les données de chaque partie.
3. On fusionne les deux parties.

La récursivité s'arrête car on finit par arriver à des listes composées d'un seul élément et le tri est alors trivial.

On peut aussi utiliser un algorithme itératif :

1. On trie les éléments deux à deux.
2. On fusionne les listes obtenues.
3. On recommence l'opération précédente jusqu'à ce qu'on ait une seule liste triée.

Complexité

- Moyenne et pire des cas : $O(n\log(n))$

```
def merge(l1, l2):
    liste = []
    i = j = 0
    n1=len(l1)
    n2=len(l2)
    while True:
        if i<n1 and j<n2:
            if l1[i]<l2[j]:
                liste.append(l1[i])
                i+=1
            else:
                liste.append(l2[j])
                j+=1
        elif i>=n1:
            liste.extend(l2[j:])
            break
        else:
            liste.extend(l1[i:])
            break
    return liste

def tri_fusion(liste):

    def tri_fusion_interne(liste):
        if len(liste)<2 :
            return liste
        return merge(tri_fusion_interne(liste[:len(liste)//2]),
                 tri_fusion_interne(liste[len(liste)//2:]))

    liste[:] = tri_fusion_interne(liste)
```

9.4.6. Tri par tas (Heapsort)

Cet algorithme permet de trier les éléments d'un tableau en $O(n \log(n))$ dans le pire des cas, où n est le nombre d'éléments à trier. Les principaux atouts de cette méthode sont la faible consommation mémoire et l'efficacité optimale.

Principe

L'idée qui sous-tend cet algorithme consiste à voir le tableau comme un arbre binaire. Le premier élément est la racine, le deuxième et le troisième sont les deux descendants du premier élément, etc. Ainsi le n-ième élément a pour fils les éléments $2n+1$ et $2n+2$. Si le tableau n'est pas de taille $2n$, les branches ne se finissent pas tout à fait à la même profondeur.

Dans l'algorithme, on cherche à obtenir un tas, c'est-à-dire un arbre binaire vérifiant les propriétés suivantes (les deux premières propriétés découlent de la manière dont on considère les éléments du tableau) :

- la différence maximale de profondeur entre deux feuilles est de 1 (i.e. toutes les feuilles se trouvent sur la dernière ou sur l'avant-dernière ligne) ;
- les feuilles de profondeur maximale sont « tassées » sur la gauche.
- chaque nœud est de valeur supérieure à celles de ses deux fils, pour un tri ascendant.

Comme expliqué au paragraphe 6.4.2, un tas ou un arbre binaire presque complet peut être stocké dans un tableau, en posant que les deux descendants de l'élément d'indice n sont les éléments d'indices $2n+1$ et $2n+2$ (pour un tableau indicé à partir de 0). En d'autres termes, les nœuds de l'arbre sont placés dans le tableau ligne par ligne, chaque ligne étant décrite de gauche à droite.

Une fois le tas de départ obtenu, l'opération de base de ce tri est le tamisage d'un élément, supposé le seul « mal placé » dans un arbre qui est presque un tas. Plus précisément, considérons un arbre $T = T[0]$ dont les deux sous-arbres ($T[1]$ et $T[2]$) sont des tas, tandis que la racine est éventuellement plus petite que ses fils. L'opération de tamisage consiste à échanger la racine avec le plus grand de ses fils, et ainsi de suite récursivement jusqu'à ce qu'elle soit à sa place.

Pour construire un tas à partir d'un arbre quelconque, on tamise les racines de chaque sous-tas, de bas en haut (par taille croissante) et de droite à gauche.

Pour trier un tableau à partir de ces opérations, on commence par le transformer en tas. On échange la racine avec le dernier élément du tableau, et on restreint le tas en ne touchant plus au dernier élément, c'est-à-dire à l'ancienne racine. On tamise la racine dans le nouveau tas, et on répète l'opération sur le tas restreint jusqu'à l'avoir vidé et remplacé par un tableau trié.

```
def faire_tas(tab, debut, n):
    # transforme le tableau "tab" en un tas
    racine = debut
    while racine*2 + 1 < n:
        fils = racine*2 + 1
        if fils < n-1 and tab[fils] < tab[fils+1]:
            fils += 1
        if tab[racine] < tab[fils]:
            tab[racine], tab[fils] = tab[fils], tab[racine]
            racine = fils
        else:
            return

def heapsort(tab):
    n = len(tab)
    debut = n//2 - 1
    fin = n - 1
    while debut >= 0:
        faire_tas(tab, debut, n)
        debut -= 1
    while fin > 0:
        tab[fin], tab[0] = tab[0], tab[fin]
        faire_tas(tab, 0, fin)
        fin -= 1
```

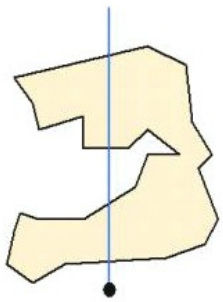
9.5. Tester si un point est dans un polygone

Dans le plan, étant donné un polygone simple (dont les arêtes ne se coupent pas), comment déterminer si un point se trouve à l'intérieur ou à l'extérieur de ce polygone ?

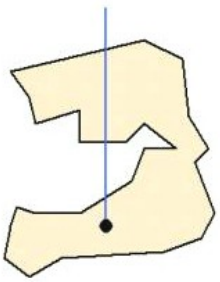
Méthode

La méthode la plus utilisée est de tracer une demi-droite de ce point vers l'infini et de compter les intersections entre cette droite et les segments du polygone.

- Tracer une demi-droite à partir du point vers l'infini.
- Calculer le nombre L d'intersections entre cette demi-droite et les côtés du polygone.
- Un nombre L impair indique que le point est à l'intérieur.
- Il faut bien sûr faire attention aux coins qui tombent juste sur la demi-droite.



Nombre pair d'intersections : point à l'extérieur



Nombre impair d'intersections : point à l'intérieur

9.5.1. Pour savoir si deux segments se coupent

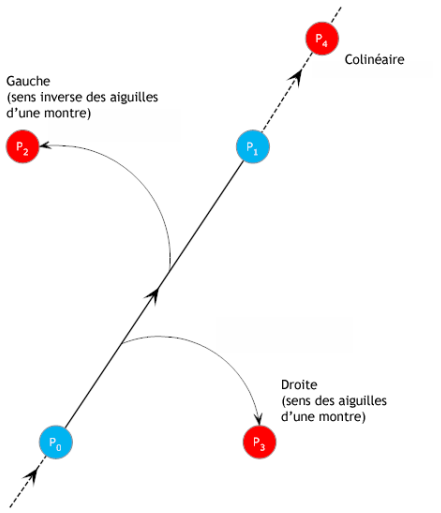
La première chose à faire est de trouver une méthode pour localiser un point P par rapport à une ligne passant par les points P_0 et P_1.

Pour différencier ces trois cas, on va utiliser les déterminants. En effet, rappelons-nous qu'un déterminant peut être interprété comme une aire signée.

Ainsi, sur le dessin ci-contre :

$\det(P_0P_1, P_0P_2) > 0$
$\det(P_0P_1, P_0P_3) < 0$
$\det(P_0P_1, P_0P_4) = 0$

où les P_iP_j désignent des vecteurs-colonnes et dét un déterminant 2 x 2.

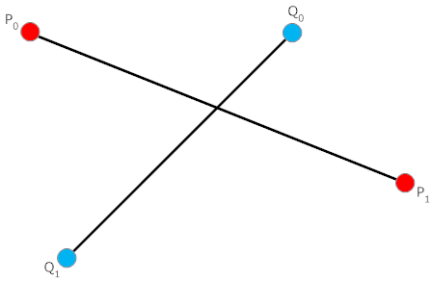


On dira que :

orientation(P_0, P_1, P_2) = 1
orientation(P_0, P_1, P_3) = –1
orientation(P_0, P_1, P_4) = 0

D'où la condition ci-dessous :

```
SI orientation(Q0,Q1,P0) ≠ orientation(Q0,Q1,P1)
ET orientation(P0,P1,Q0) ≠ orientation(P0,P1,Q1) ALORS
    RETOURNER « les deux segments se coupent »
```

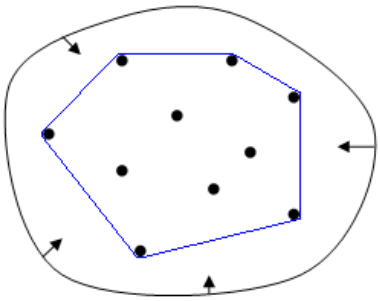


Exercice 9.8

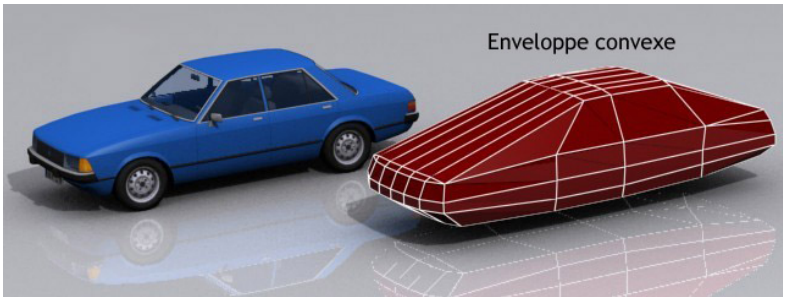
Écrivez un programme Python qui implémente la méthode vue ci-dessus. Le polygone sera donné par la liste de ses sommets. Vous trouverez sur le site compagnon une ébauche à compléter.

9.6. Enveloppe convexe

Imaginons une planche avec des clous qui dépassent. Englobons ces points avec un élastique que l'on relâche. Le polygone obtenu (en bleu ci-dessous) est l'enveloppe convexe (*convex hull*).



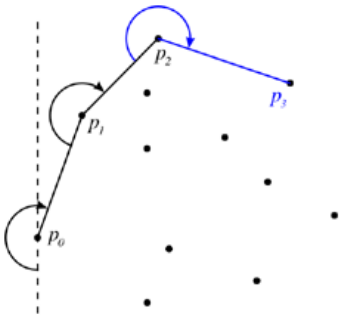
En trois dimensions, l'idée serait la même avec un ballon qui se dégonflerait jusqu'à être en contact avec tous les points qui sont à la surface de l'enveloppe convexe.



9.6.1. Marche de Jarvis (Gift wrapping algorithm)

La marche de Jarvis est un algorithme qui « enveloppe » un ensemble de points dans un « papier cadeau » : on accroche ce papier à un point initial p_1, puis on le tend, et on tourne autour du nuage de points. Le premier point rencontré par le papier sera p_1, puis p_2, ... jusqu'à retrouver p_0.

On construit donc l'enveloppe convexe segment par segment, en partant du point p_0. Il ne doit y avoir aucun point à gauche du prochain segment $p_i p_{i+1}$. Il n'y a qu'un point p_{i+1} qui satisfait cette condition ; il faut le chercher parmi les points qui ne sont pas encore sur l'enveloppe convexe. On s'arrête lorsque $p_{i+1} = p_0$.



Remarquons que les points ne sont pas triés.

Cet algorithme doit son nom à R. A. **Jarvis**, qui publia cet algorithme en 1973.

Complexité

La complexité est en $O(nh)$, où n est le nombre total de sommets et où h représente le nombre de sommets de l'enveloppe convexe. On qualifie ce genre d'algorithme de « sensible à la sortie ».

Exercice 9.9

Écrivez un programme Python qui implémente la marche de Jarvis. Vous trouverez sur le site compagnon une ébauche à compléter.

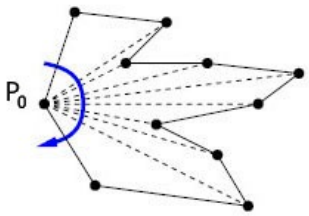
9.6.2. Parcours de Graham (Graham's scan)

Ronald **Graham**
(1935 - 2020)

Cet algorithme doit son nom à Ronald **Graham**, qui a publié l'algorithme original en 1972.

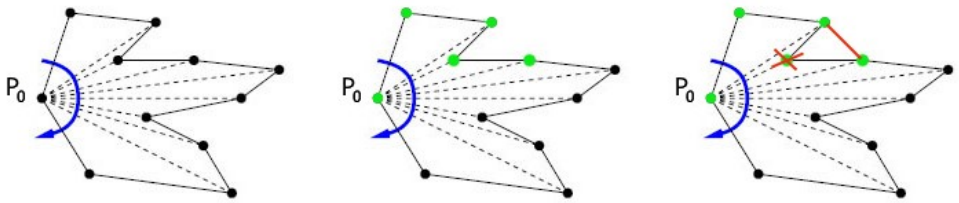
La première étape de cet algorithme consiste à rechercher le point le plus à gauche. S'il y a égalité entre plusieurs points, l'algorithme choisit parmi eux le point de plus petite ordonnée. Nommons P_0 ce point. La complexité en temps de cette étape est en $O(n)$, n étant le nombre de points de l'ensemble.

Les autres points P_i sont ensuite triés en fonction de la pente du segment P_0P_i, de la plus grande pente à la plus petite. N'importe quel algorithme efficace de tri convient pour cela. À l'issue de cette étape, on dispose d'un tableau T contenant les points ainsi triés. P_0 sera le premier élément de ce tableau.

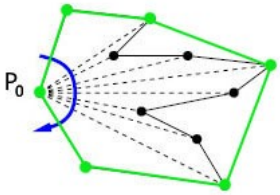


L'algorithme considère ensuite successivement les séquences de trois points contigus dans le tableau T, vus comme deux segments successifs. On regarde ensuite si ces deux segments mis bout à bout constitue un « tournant à gauche » ou un « tournant à droite ».

- Si l'on rencontre un « tournant à droite », l'algorithme passe au point suivant de T.
- Si c'est un « tournant à gauche », cela signifie que l'avant-dernier point considéré (le deuxième des trois) ne fait pas partie de l'enveloppe convexe, et qu'il doit être enlevé de T. Cette analyse se répète ensuite, tant que l'ensemble des trois derniers points est un « tournant à gauche » .



Le processus se terminera quand on retombera sur le point P_0. T contiendra alors les points formant l'enveloppe convexe.



Complexité

Le tri des points peut se faire avec une complexité en temps en $O(n\log(n))$. La complexité de la boucle principale peut sembler être en $O(n^2)$, parce que l'algorithme revient en arrière à chaque point pour évaluer si l'un des points précédents est un « tournant à droite ». Mais elle est en fait en $O(n)$, parce que chaque point n'est considéré qu'une seule fois. Ainsi, chaque point analysé soit termine la sous-boucle, soit est retiré de T et n'est donc plus jamais considéré. La complexité globale de l'algorithme est donc en $O(n\log(n))$, puisque la complexité du tri domine la complexité du calcul effectif de l'enveloppe convexe.

Exercice 9.10

Écrivez un programme Python qui implémente le parcours de Graham. Vous trouverez sur le site compagnon une ébauche à compléter.

9.6.3. L'algorithme de Chan

L'algorithme de Chan nommé d'après son inventeur Timothy M. **Chan**, est un algorithme sensible à la sortie qui calcule l'enveloppe convexe d'un ensemble P de n points, en dimension 2 ou 3. En dimension 2, l'algorithme combine un algorithme en $O(n\log(n))$ (par exemple le parcours de Graham) et la marche de Jarvis afin d'obtenir un algorithme en $O(n\log(h))$, où h est le nombre de points dans l'enveloppe convexe.

Petit problème : h n'est pas connu…

Phase 1 : pré-calcul de sous-enveloppes convexes

On suppose ici que $m = h$ (inconnu). Nous verrons plus loin comment choisir m.

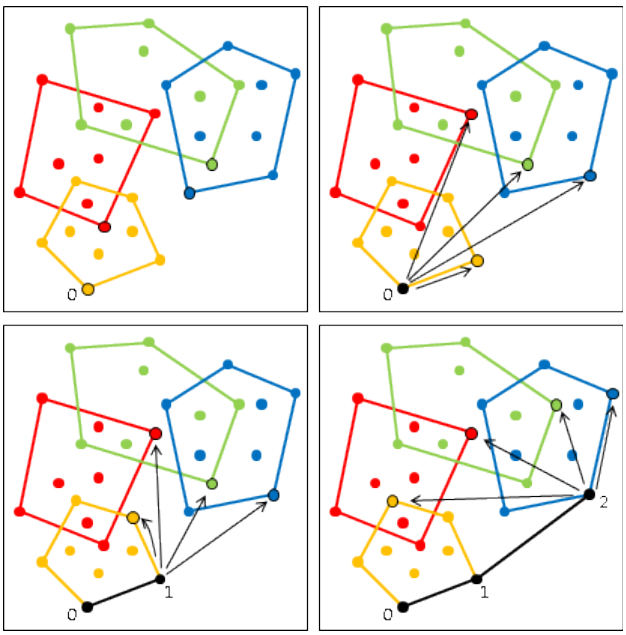
L'algorithme commence par partitionner P en au plus n/m sous-ensembles, avec au plus m points dans chaque sous-ensemble. Puis on calcule l'enveloppe convexe de chacun des sous-ensembles en utilisant un algorithme en $O(n\log(n))$, par exemple le parcours de Graham.

Phase 2 : calcul de l'enveloppe convexe

La seconde phase consiste à exécuter une variante de marche de Jarvis en utilisant les sous-enveloppes convexes pré-calculées dans la phase 1.

1. Commencer par un point extrême p_0 (on a choisi p_0 le point le plus bas dans l'illustration ci-dessous). On sait que ce point appartient à l'enveloppe convexe. Poser $i = 0$.
2. Pour chacune des sous-enveloppes convexes, trouver le point q_k de sorte que tous les points de la sous-enveloppe k soient à gauche du segment orienté p_iq_k.
3. Parmi ces segments, garder celui qui n'a aucun des autres points q_j à sa droite. C'est un bord de l'enveloppe convexe. Appeler q l'extrémité terminale de ce segment orienté.
4. Poser $i := i + 1$ et $p_i := q$. Retourner en 2 tant $p_i \neq p_0$ et $i < m$.
5. Si $p_i = p_0$, l'enveloppe convexe est la suite des points $[p_0, p_1, \ldots, p_i]$, sinon recommencer la phase 1 avec un m plus grand.

Timothy M. **Chan**
(né en 1976)



Notons qu'il existe encore d'autres algorithmes pour calculer l'enveloppe convexe d'un nuage de points.
Citons notamment : Quickhull, l'algorithme de Shamos et l'algorithme de Kirkpatrick et Seidel.

Quelle valeur prendre pour *m* ?

L'algorithme décrit ci-dessus utilise $m = h$. Si $m < h$, nous nous en rendons compte dans la marche de Jarvis au bout de $m+1$ étapes. Ainsi, si $m < h$, on ne calcule pas l'enveloppe convexe jusqu'au bout. Si $m > h$, l'algorithme s'arrête et calcule l'enveloppe convexe complète.

L'idée consiste alors à commencer l'algorithme avec une valeur petite pour m (dans l'analyse qui suit on utilise 2, mais des nombres aux alentours de 5 marchent mieux en pratique), puis on augmente la valeur de m jusqu'à ce que $m > h$.

Si on augmente la valeur de m trop lentement, on a besoin de répéter les phases 1 et 2 trop de fois et le temps de d'exécution est trop grand. *A contrario*, si on augmente les valeurs de m trop vite, on risque d'atteindre une valeur de m beaucoup trop grande par rapport à h, et le temps d'exécution est aussi trop grand.

À l'itération t (on commence à 1), m prend la valeur $\min(n, 2^{2^t})$. En d'autres termes, m prend les valeurs 4, 16, 256, …, sans dépasser n.

Sources

[1] Wikipédia, « Algorithmique », <http://fr.wikipedia.org/wiki/Algorithmique>

[2] Wikipédia, « Algorithmes de tri », <http://fr.wikipedia.org/wiki/Catégorie:Algorithme_de_tri>

[3] Wikipédia, « Marche de Jarvis », <http://fr.wikipedia.org/wiki/Marche_de_Jarvis>

[4] Wikipédia, « Parcours de Graham », <http://fr.wikipedia.org/wiki/Parcours_de_Graham>

[5] Wikipédia, « Algorithme de Chan », <http://fr.wikipedia.org/wiki/Algorithme_de_Chan>