Algorithmes à connaître

Les algorithmes présentés ici sont à connaître :
- ils font partis de la culture de base en informatique ;
- par de petites modifications, ils permettent de résoudre de nombreux problèmes.

Table des matières

1 Algorithmes classiques

1.1 Division euclidienne

```python
def quotient_reste(a,b):
    """
    Renvoie le quotient q et le reste r de la division de a par b
    Arguments:
        a, int : dividende, entier naturel
        b, int : diviseur, entier naturel
    Retour:
        q, int : quotient
        r, int : reste
    """
        r, q = a, 0
        while r >= b:
            r = r - b
            q = q + 1
        return(q,r)
```

1.2 Calcul de puissance

1.2.1 Algorithme naïf

```python
    def exponentiation_naive(x,n):
        """
        Renvoie x**n par la methode naive.
        Arguments :
            x, flt : réel
            n, int : entier naturel
        Retour :
            res,flt : resultat
        """
        res = 1
        while n>=1:
            res = res * x
            n=n-1
        return(res)
```

1.2.2 Exponentiation rapide

```python
    def exponentiation_rapide(x,n):
        """
        Renvoie x**n par la methode d'exponentiation rapide.
        Arguments:
            x, flt : un nombre réel
            n, int : un nombre entier naturel
        Retour :
            res,flt : resultat
        """
        res = 1
        a = x
```

```python
        while n>0:
            if n%2 == 1:
                res=res*a
            a=a*a
            n=n//2
        return(res)
```

1.3 Conversion décimale ⇆ binaire

1.3.1 Conversion décimale → binaire

```python
def Conversion_decimale_binaire(n):
    """
    Renvoie l'expression binaire d'un entier positif n
    Argument:
        n, int : entier naturel
    Retour :
        S,str : expression binaire
    """
    S=''
    while n>0:
        S=str(n%2)+S
        n=n//2
    return(S)
```

1.3.2 Conversion décimale ← binaire

```python
def Conversion_binaire_decimale(S):
    """
    Renvoie un entier positif correspondant àune expression binaire
    Keyword arguments:
    Argument :
        S,str : expression binaire
    Retour :
        n, int : un nombre entier naturel
    """
    N=0 #initialisation
    for i in range(len(S)):
        N=N+int(S[len(S)-1-i])*2**i
    return(N)
```

2 Recherches dans une liste

2.1 Recherche d'un nombre dans une liste

```python
def is_number_in_list(nb,L):
    """Renvoie True si le nombre entier nb est dans la liste de nombres L
    Arguments:
        nb, int : nombre entier
        L, list : liste de nombres entiers
    Retour:
        bool : True or False
    """
    for i in range(len(L)):
        if L[i]==nb:
            return(True)
    return(False)
```

2.2 Recherche du maximum dans une liste de nombre

```python
def what_is_max(L):
    """
    Renvoie le plus grand nombre d'une liste
    Arguments :
        L: liste de nombres
    Retour :
        maxi, flt ou int : maximum de la liste
    """
    maxi=L[0] # ou maxi=-float('inf')
    for i in range(len(L)):
        if L[i]>maxi:
            maxi=L[i]
    return(maxi)
```

2.3 Recherche du maximum dans un tableau

Un tableau est codé en machine par une liste de listes de nombres.

```python
def what_is_max(T):
    """
    Renvoie le plus grand nombre d'un tableau T
    Arguments :
        T:  liste de listes de nombres
    Retour :
        maxi, flt ou int : maximum de la liste
    """
    maxi=T[0][0] # ou maxi=-float('inf')
    for i in range(len(T)):
        for j in range(len(T[i])
            if T[i][j]>maxi:
                maxi=T[i]
    return(maxi)
```

2.4 Recherche par dichotomie dans une liste triée

```python
def is_number_in_list_dicho(nb,L):
    """
    Recherche d'un nombre par dichotomie dans une liste triée par ordre CROISSANT.
    Renvoie l'index si le nombre nb est dans la liste de nombres L.
    Renvoie None sinon.
    Arguments :
        nb, int ou flt : nombre
        L, list : liste de nombres entiers triés
    Retour :
        None
        ou
        m, int: index entier
    """
    g, d = 0, len(L)-1
    while g <= d:
        m = (g + d) // 2
        if L[m] == nb:
            return(m)
        if L[m] < nb:
            g = m+1
        else:
            d = m-1
    return(None)
```

3 Traitement d'une liste de nombres

3.1 Calcul de la moyenne

```python
def calcul_moyenne(L):
    """
    Renvoie la moyenne des valeurs d'une liste de
    nombres.
    Argument :
        L, flt : liste de nombres
    Retour :
        flt : moyenne
    """
    res = 0
    for i in range(len(L)):
        res = res+L[i]
    return(res/len(L))
```

3.2 Calcul de la variance

Soit une série statistique prenant les n valeurs x_1, x_2, ..., x_n. Soit m la moyenne de ces valeurs. La variance est définie par :

$$v = \frac{1}{n}\sum_{i=1}^{n}(x_i - m)^2$$

```python
def calcul_variance(L,m):
    """
    Renvoie la variance des valeurs d'un tableau.
    Argument :
        L, list : liste de nombres
    Retour :
        flt : la variance
    Nécessite la fonction calcul_moyenne
    """
    m=calcul_moyenne(L)
    res = 0
    for i in range(len(L)):
        res = res+(L[i]-m)**2
    return(res/len(L))
```

4 Algorithmes par boucles imbriquées

4.1 Deux valeurs les plus proches dans une liste

```python
def rech_2proches(L):
    """ Recherche les deux valeurs les plus proche dans une liste.
    Renvoie les index de ces deux valeur.
    Arguments:
        L,list : liste
    Retour:
        tuple : le tuple formé par les duex index
    """
    n = len(L)
    ind = [0 , 1]
    d = abs( L[0]-L[1])
    for i in range(n-1):
        for j in range (i+1,n):
            dij = abs(L[i]-L[j])
            if dij < d :
                d = dij
                ind = [ i , j ]
    return(ind)
```

4.2 Recherche d'un mot dans une chaîne de caractères

```python
def index_of_word_in_text(mot, texte):
    """ Recherche si le mot est dans le texte.
    Renvoie l'index si le mot est présent, None sinon.
    Arguments:
        mot,str : mot recherché
        texte, str : texte complet
    Retour:
        None ou i,int : index de la première lettre du mot dans le texte
    """
```

```python
    for i in range(1 + len(texte) - len(mot)):
        j = 0
        while j < len(mot) and mot[j] == texte[i + j]:
            j += 1
        if j == len(mot):
            return(i)
    return(None)
```

5 Algorithmes de tri

5.1 Tris itératifs quadratiques

5.1.1 Tri à bulles

```python
def triABulles(L ):
    n = len(L)
    for i in range(n-1):
        for j in range(0, n-i-1):
            if L[j] > L[j+1] :
                L[j], L[j+1] = L[j+1], L[j]
```

5.1.2 Tri par selection

```python
def rang_mini_from_j(L, j):
    i, m = j, L[j]
    for k in range(j+1, len(L)):
        if L[k] < m:
            i, m = k, L[k]
    return(i)

def Tri_selection(L):
    for j in range(len(L)-1):
        i = rang_mini_from_j(L, j)
        if i != j:
            L[i], L[j] = L[j], L[i]
```

5.1.3 Tri par insertion

```python
def triInsertion(L):
    for i in range(1, len(L)):
        k = i
        v = L[i]
        while (k > 0) and (v < L[k-1]):
            L[k] = L[k-1]
            k = k-1
        L[k] = v
```

5.2 Tris récursifs efficients

5.2.1 Tri par partition-fusion

```python
def fusion(L1, L2):
    """fusionne deux listes triées"""
    if L1 == []:
        return L2
    if L2 == []:
        return L1
    #Désormais, les deux listes sont non vides.
    if L1[0] <= L2[0]:
        return([L1[0]] + fusion(L1[1:], L2))
    else:
        return([L2[0]] + fusion(L1, L2[1:]))


def TriFusion(L):
    if len(L) <= 1:
        return L
    m = len(L)//2
    return(fusion(TriFusion(L[:m]), TriFusion(L[m:])))
```

5.2.2 Tri rapide *quick sort*

```python
def quicksort(L):
    if len(L) <= 1:
        return L
    else:
        pivot = L[0]
        plus_grand = []
        plus_petit = []
        for x in L[1:]:
            if x >= pivot:
                plus_grand.append(x)
            else:
                plus_petit.append(x)
        return(quicksort(plus_petit) + [pivot] + quicksort(plus_grand))
```

5.3 Autre tri : tri par comptage

```python
def comptage (L,n) :
    P = [0 for i in range(n)]
    for k in L :
        P[k] += 1
    return(P)


def TriComptage (L,N) :
    M = []
    P = comptage(L,N)
    for k in range(N) :
        for i in range(P[k]) :
            M.append(k)
    return(M)
```

6 Algorithmes d'analyse de graphes

Un graphe est modélisé en machine par une liste, une matrice ou un dictionnaire d'adjacence. (ce graphe peut être orienté ou non, pondéré ou non)

6.1 Parcours en largeur : itératif

L'algorithme explore le graphe à partir d'un nœud `depart`, en explorant les nœuds les plus proches en priorité. Il renvoit la liste des nœuds découverts, dans l'ordre de découverte. Il s'appuie sur une **file**.

Algorithme 1 : Algorithme de parcours de graphe en largeur : Version 1

```
1  Initialisation;
2  Initialiser la liste file avec le nœud départ ;
3  Initialiser la liste nœuds_connus avec le nœud départ ;
4  tant que file n'est pas vide faire
5      nœud_courant ← Défiler(file) ;
6      pour chacun des voisins v de nœud_courant faire
7          si v n'appartient pas à la liste nœuds_connus alors
8              Ajouter v à la liste nœuds_connus;
9              Enfiler v à la fin de la liste file;
10         fin
11     fin
12 fin
13 return nœuds_connus
```

6.2 Parcours en profondeur : itératif

L'algorithme explore le graphe à partir d'un nœud `depart`, en choisissant en priorité le nœud connu le plus éloigné du nœud `depart`. Il renvoit la liste des nœuds découverts, dans l'ordre de découverte. Il s'appuie sur une **pile**.

Algorithme 2 : Algorithme de parcours de graphe en profondeur itératif

```
1  Initialisation;
2  Initialiser la pile avec le nœud de départ;
3  Initialiser la liste nœuds_connus;
4  tant que la pile n'est pas vide faire
5  |  nœud courant ← Dépiler(pile) ;
6  |  si il existe un voisin du nœud courant non connu alors
7  |  |  Ajouter le voisin à la liste nœuds_connus ;
8  |  |  Empiler le nœud courant ;
9  |  |  Empiler ce voisin ;
10 |  |  sinon
11 |  |  |  Ajouter le nœud courant à la liste exploré
12 |  |  fin
13 |  fin
14 fin
15 return nœuds_connus
```

6.3 Parcours en profondeur : récursif

L'algorithme explore le graphe à partir d'un nœud `depart`, en choisissant les nœuds les plus éloignés du nœud `depart`. Il s'appuie sur une fonction recursive `Visiter`, et une variable globale `n\oe{}uds_connus`.

Algorithme 3 : Algorithme de parcours de graphe en profondeur récursif

```
1  Initialisation;
2  Initialiser la liste nœuds_connus avec le nœud départ ;
3  Visiter(nœud_depart,nœuds_connus);
4  return nœuds_connus
```

Algorithme 4 : Fonction récursive Visiter(u,nœuds_connus)

```
1  si u n'est pas dans la liste nœuds_connus alors
2  |  Ajouter u à la liste nœuds_connus;
3  fin
4  pour chaque sommet v voisin de u non connu faire
5  |  Visiter(v,nœuds_connus);
6  fin
```

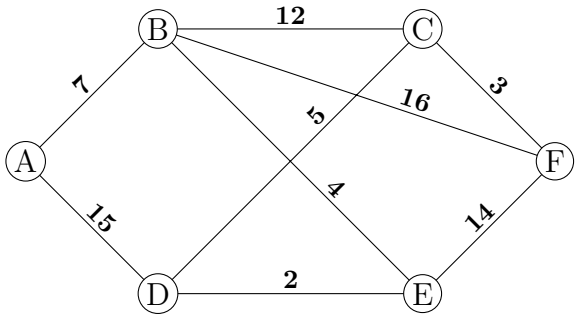
6.4 Chemin le plus court : algorirhme de Dijkstra

L'algorithme de Dijkstra détermine tous les chemins les plus courts à partir d'un nœud de départ vers tous les autres nœuds. Les distances entre les nœuds doivent être **positives**.

Algorithme 5 : Algorithme de Dijkstra

```
1  Initialisation;
2  Initialiser le tableau avec 3 informations : pour chaque sommet :
3          la distance au sommet de départ (∞ car inconnue à l'initialisation)
4          son sommet prédécesseur dans le chemin (∅ car inconnu à l'initialisation) ;
5  nœud_courant ← le sommet de départ ;
6  tant que tous les sommets n'ont pas été un nœud courant (ou traité) faire
7      pour chacun des voisins v de nœud_courant faire
8          si la distance en passant par nœud_courant est strictement plus courte que
             celle indiquée dans le tableau alors
9              Mettre à jour le tableau avec cette nouvelle distance et comme
                 prédécesseur nœud_courant
10         fin
11         ;
12     fin
13     nœud_courant ← sommet non traité le plus proche du sommet de départ;
14 fin
15 return tableau
```

Exemple :




```
L=[[A, [[B,7], [D,15]]],
   [B, [[C,12], [E,4], [F,16]]],
   [C, [[D,5], [F,3]]],
   [D, [[E,2]]],
   [E, [[F,14]]]]
```

Déroulé de l'algorithme :

A	B	C	D	E	F
0	∞	∞	∞	∞	∞
—	$\mathbf{7_A}$	∞	15_A	∞	∞
—	—	19_B	15_A	$\mathbf{11_B}$	23_B
—	—	19_B	$\mathbf{13_E}$	—	23_B
—	—	$\mathbf{18_D}$	—	—	23_B
—	—	—	—	—	$\mathbf{21_C}$

On peut en déduire la distance A-F : 21.

En utilisant les informations des prédécesseurs, on peut reconstruire le chemin en remontant à partir de la destination : A-B-E-D-C-F.