Boîte à outils et rappels

Prédicats sur les listes

Pour montrer $\exists x \in \mathrm{L}, \mathscr{P}(x)$:

Au moins un élément de la liste L est strictement négatif :

```python
def au_moins_un_strictement_negatif(L) :
    for x in L :
        if x < 0 :          # Dès le premier élément négatif
            return True     # on retourne True

    return False            # Aucun x n'est négatif
                            # on retourne False
```

Solution Python (souvent à éviter aux concours) :

```python
def au_moins_un_strictement_negatif(L) :
    return any( x<0 for x in L )
```

Prédicats sur les listes

Pour montrer $\forall x \in \mathrm{L}, \mathscr{P}(x)$ on écrit généralement $\nexists x \in \mathrm{L}, \neg\mathscr{P}(x)$:

Tous les éléments de la liste L sont positifs ou nuls :

```python
def tous_positifs_ou_nuls(L) :
    for x in L :
        if x < 0 :          # Dès le premier élément négatif
            return False    # on retourne False

    return True             # Aucun x n'est négatif
                            # on peut retourner True
```

Solution Python (souvent à éviter aux concours) :

```python
def tous_positifs_ou_nuls(L) :
    return all( x>=0 for x in L )
```

Invariants de boucle

Invariant : propriété qui reste vraie à chaque itération de la boucle

Par exemple : « il y a une racine dans [a, b] »

On justifie un invariant par récurrence, en montrant que :

- il est vrai lors de la première itération
- s'il est vrai à l'itération $k \geqslant 0$, alors il l'est aussi à l'itération $k+1$

Ne pas hésiter à mettre les invariants en commentaire !

Invariants de boucle

On peut avoir à préciser à quel endroit de la boucle la propriété est vérifiée

« au début de l'itération, le nom u désigne u_k »

« à la fin de l'itération, le nom u désigne u_{k+1} »

```
def U(n) :
    u = 1.0          # u désigne u_0 (u_0 = 1)

    for k in range(n) :
        # Invariant : u désigne u_k

        u = 3.14 * u * (1.0-u)   # passage de u_k à u_{k+1}

        # Invariant : u désigne u_{k+1}

    return u         # u désigne u_n
```

Premier élément qui vérifie $\mathscr{P}(x)$

On parcourt les éléments, **return** quand on trouve

Index et valeur du premier élément positif ou nul :

```python
def premier_positif(x, L) :
    for i in range(len(L)) :
        if L[i] >= 0 :
            return i, L[i]
```

Variante (possiblement non permise aux écrits) :

```python
def premier_positif(x, L) :
    for i, elem in enumerate(L) :
        if elem >= 0 :
            return i, elem
```

Les fonctions retournent ici **None** si rien ne convient.

Dernier élément qui vérifie $\mathscr{P}(x)$

Même chose, mais en itérant la liste *à l'envers*

Index et valeur du dernier élément positif :

```python
def dernier_positif(x, L) :
    for i in range(len(L)-1, -1, -1) :
        if L[i] >= 0 :
            return i, L[i]
```

Variante (très possiblement non permise aux écrits) :

```python
def dernier_positif(x, L) :
    for i, elem in reversed(enumerate(L)) :
        if elem >= 0 :
            return i, elem
```

Les fonctions retournent ici **None** si rien ne convient.

Plus grand élément, plus petit élément...

On utilise des noms à fin de mémoriser ce que l'on a vu !

Invariant – Au *début* de la boucle :

`maxi` contient le plus grand élément parmi `L[:i]`

`imax` contient l'index du plus grand élément parmi `L[:i]`

```python
def maximum(L) :
    imax = 0
    maxi = L[0]

    for i in range(1, len(L)) :
        if L[i] > maxi :
            imax = i
            maxi = L[i]

    return imax, maxi
```

Dernier élément qui vérifie $\mathscr{P}(x)$, le retour

Si on ne peut pas itérer *à l'envers* (ex : fichiers)...

Là encore, on utilise un nom pour mémoriser les éléments :

Invariant – Au *début* de la boucle, `candidat` désigne le couple (index, valeur) du dernier élément vérifiant $\mathscr{P}(x)$ dans `L[:i]`

```python
def dernier_positif(L) :
    candidat = None

    for i in range(len(L)) :
        if L[i] >= 0 :
            candidat = i, L[i]

    return candidat
```

La fonction retourne ici **None** si rien ne convient.

Remplacer L*k

`[1, 3.7, "A"]*2` ↦ `[1, 3.7, "A", 1, 3.7, "A"]`

```python
def repete(L, k): # remplace L*k
    res = []
    for i in range(k):
        for j in range(len(L)):
            res.append(L[j])
    return res
```

Remplacer L1+L2 et L1.extend(L2)

```python
def concat(L1, L2): # remplace L1+L2
    res = []
    for i in range(len(L1)):
        res.append(L1[i])
    for i in range(len(L2)):
        res.append(L2[i])
    return res
```

```python
def etend(L1, L2): # remplace L1.extend(L2)
    for i in range(len(L2)):
        L1.append(L2[i])
```

Remplacer `L.index(elem)`

```python
def position(L, elem):
    for i in range(len(L)):
        if L[i] == elem:
            return i

    # Action si on ne le trouve pas
    return -1
```

Remplacer L.pop(k)

```python
def extrait(L, k):
    res = L[k]
    for i in range(k, len(L)-1):
        L[i] = L[i+1]
    L.pop()
    return res
```

Pour un équivalent à `L.remove(elem)`, on combinera les deux fonctions précédentes (on détermine la position, puis on retire l'élément)

Remplacer les compréhensions

Pour écrire `L = [ expr for i in range(n) ]`

```
L = []
for i in range(n):
    L.append(expr)
```

Pour écrire `L = [ expr for i in range(n) if cond ]`

```
L = []
for i in range(n):
    if cond:
        L.append(expr)
```

Remplacer `L.insert(k, elem)`

```python
def insertion(L, k, elem):
    L.append(L[-1])
    for i in range(len(L)-2, k, -1):
        L[i] = L[i-1]
    L[place] = elem
```

Pour une insertion dans une *nouvelle* liste :

```python
def copie_et_insertion(L, k, elem):
    res = []
    for i in range(k):
        res.append(L[i])
    res.append(elem)
    for i in range(k, len(L):
        res.append(L[i])
    return res
```

Remplacer `L.copy()`

```python
def copie(L) :
    res = []
    for i in range(len(L)):
        res.append(L[i])
    return res
```

Remplacer `L.reverse()`

```python
def retourne(L):
    n = len(L)
    for i in range(n//2):
        L[i], L[n-1-i] = L[n-1-i], L[i]
```

Pour une *nouvelle* liste retournée :

```python
def copie_et_retourne(L):
    res = []
    for i in range(len(L)-1, -1, -1):
        res.append(L[i])
    return res
```

Remplacer all(L) et any(L)

```python
def tous(L): # remplace all(L)
    for i in range(len(L)):
        if not L[i]:
            return False
    return True
```

```python
def existe(L): # remplace any(L)
    for i in range(len(L)):
        if L[i]:
            return True
    return False
```

Filtrer une liste

S'il faut filtrer une liste (ne conserver que les éléments vérifiant un critère), il vaut toujours mieux créer une *nouvelle* liste.

```python
def retire_zeros(L):
    res = []
    for elem in L:
        if elem != 0:
            res.append(elem)
    return res
```

Filtrer une liste

Si **vraiment** il faut le faire en place, on peut réécrire sur le début de la liste et éliminer ce qui n'a pas été réécrit. **On ne retire ni ajoute aucun élément durant l'itération** !

```python
def retire_zeros_en_place(L):
    i = 0
    for elem in L:
        if elem != 0:
            L[i] = elem
            i = i+1

    for _ in range(i, len(L)):
        L.pop()
```

Recherche dichotomique

Cette fonction est-elle correcte ?

```python
def recherche_dicho(L, x) :
    """Cherche x dans L croissante par dichotomie"""
    g, d = 0, len(L)-1

    while g < d :
        m = (g+d)//2
        if L[m] == x :
            return True
        if L[m] < x :
            g = m
        else :
            d = m

    return False
```

Non : boucle infinie sur `L = [ 0, 1 ]` et `x = 2` :

```python
def recherche_dicho(L, x) :
    """Cherche x dans L croissante par dichotomie"""
    g, d = 0, len(L)-1      # g = 0, d = 1

    while g < d :           # 0 < 1 est vrai
        m = (g+d)//2        # m = 0
        if L[m] == x :      # L[0] = 0, x = 2 donc faux
            return True
        if L[m] < x :       # L[0] = 0, x = 2 dond vrai
            g = m           # -> g = 0, rien ne change!
        else :
            d = m

    return False
```

Recherche dichotomique

Première étape : choisir un invariant de boucle (et l'écrire) :

On recherche x à une position `i` ∈ [g, d [

On recherche x à une position `i` ∈ [g, d]

On recherche x à une position `i` ∈] g, d [

On recherche x à une position `i` ∈] g, d]

Le choix est libre, quoiqu'on préférera l'un des deux premiers

(car on n'a pas forcément $\lfloor \frac{g+d}{2} \rfloor > g$)

Mais il faut faire un choix clair, et :

- choisir les valeurs initiales de g et d en conséquence

  `g=0` ou `g=-1`, `d=len(L)` ou `d=len(L)-1`

- adapter la condition du **`while`**

  `g<=d`, `g<d`, `g<d+1`...

Recherche dichotomique

Ensuite, pour garantir la terminaison, il faut qu'à chaque itération :

- soit l'intervalle est *strictement* réduit
- soit on sort de la boucle **`while`** (**`return`** ou condition non vraie)

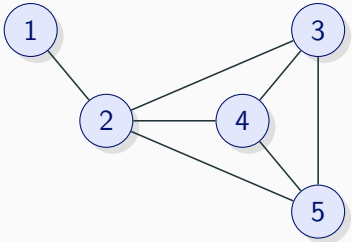
Principal souci : on n'a pas $g < \lfloor \frac{g+d}{2} \rfloor$ si $d = g+1$!

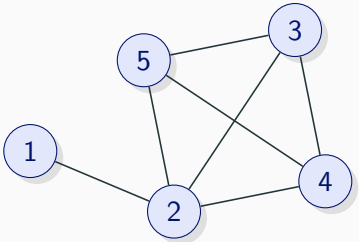
Si on écrit `g=m`, a-t-on toujours préalablement `m>g` ?

Si on écrit `g=m+1`, n'ignore-t-on pas de valeur ?

Un ensemble de sommets V et un ensemble d'arêtes $V \subseteq E^2$

$$G \begin{cases} V = \{1,2,3,4,5\} \\ E = \{(1,2),(2,3),(2,4),(2,5),(3,4),(3,5),(4,5)\} \end{cases}$$

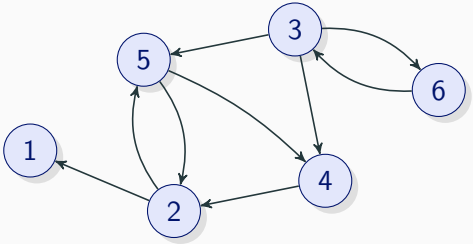




Variantes

On peut :

- associer des données aux sommets
- associer des données aux arêtes
- orienter les arêtes (graphes orientés, arêtes $\mapsto$ arcs)



Représentation

Énormément de représentations possibles !

Principalement :

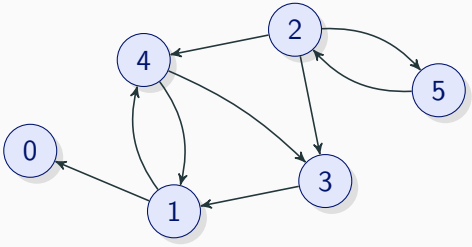
- matrices (liste de listes) d'adjacence

  `M[i][j] == True` (ou `1`) ssi il y a un arc de i à j

- liste/dictionnaire de listes des voisins

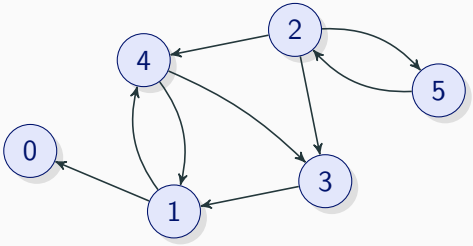
  `M[i]` contient la liste des voisins de j

Variantes



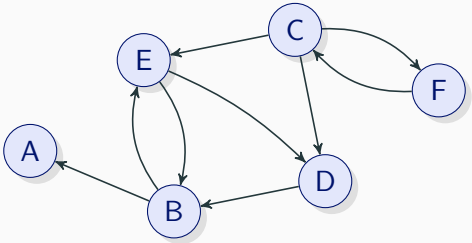

```
[ [ 0, 0, 0, 0, 0, 0 ],
  [ 1, 0, 0, 0, 1, 0 ],
  [ 0, 0, 0, 1, 1, 1 ],
  [ 0, 1, 0, 0, 0, 0 ],
  [ 0, 1, 0, 1, 0, 0 ],
  [ 0, 0, 1, 0, 0, 0 ] ]
```

Variantes



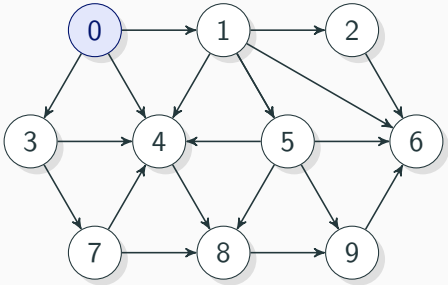

```
[ [],
  [0, 4],
  [3, 4, 5],
  [1],
  [1, 3],
  [2] ]
```

Variantes




```
{ "A": [],
  "B": ["A", "E"],
  "C": ["D", "E", "F"],
  "D": ["B"],
  "E": ["B", "D"],
  "F": ["C"] }
```

Parcours de graphes



Parcours en largeur

```python
def explore(graphe, source):
    L = [source]
    visites = {source: None}

    while L != []:
        next = []
        for s in L:
            for v in voisins(s, graphe):
                if v not in visites:
                    next.append(v)
                    visites[v] = s
        L = next

    return visites
```

Parcours en largeur

```python
def explore(graphe, source):
    L = [source]
    visites = {source: None}

    while L != []:
        s = L.pop(0)
        for v in voisins(s, graphe):
            if v not in visites:
                L.append(v)
                visites[v] = s

    return visites
```

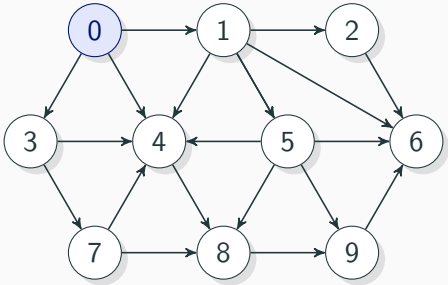
Parcours en largeur

```python
def explore(graphe, source):
    D = deque()
    D.append(source)
    visites = {source: None}

    while D:
        s = D.popleft()
        for v in voisins(s, graphe):
            if v not in visites:
                D.append(v)
                visites[v] = s

    return visites
```

Parcours



Parcours en profondeur

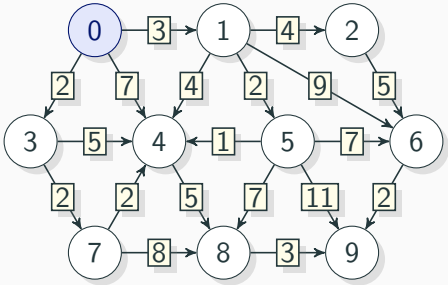
```python
def explore(graphe, source):
    visites = {source: None}

    def traite(s):
        for v in voisins(s, graphe):
            if v not in visites:
                visites[v] = s
                traite(v)

    traite(source)

    return visites
```

Plus court chemin



Dijkstra

```python
def dijkstra(graphe, source):
    visites = {}
    ouverts = { source: (None, 0) }

    while ouverts != {}:
        best = None
        for s in ouverts:
            if best is None or ouverts[s][1] < ouverts[best][1]:
                best = s
        visites[best] = ouverts[best]
        del ouverts[best]

        for v, cout in voisins(s, graphe):
            if v not in visites and (v not in ouverts
                  or ouverts[v][1] > visites[best][1]+cout):
                ouverts[v] = (best, visites[best][1]+cout)

    return visites
```

Dijkstra

Coût en $O(n^2)$ (n nombre de sommets)

Avec une file de priorité, $O(p \log n)$ (p nombre d'arcs)

$$u_n = u_{n-2} + u_{n-3}, \quad u_0 = u_1 = u_2 = 1$$

Implémentation naïve :

```python
def u(n):
    if n<=2:
        return 1
    return u(n-2) + u(n-3)
```

Coût rapidement exhorbitant

$$u_n = u_{n-2} + u_{n-3}, \quad u_0 = u_1 = u_2 = 1$$

Par remplissage de tableau

```python
def u(n):
    U = [1, 1, 1]
    for i in range(3, n+1):
        U.append(U[i-2] + U[i-3])
    return U[n]
```

$$u_n = u_{n-2} + u_{n-3}, \quad u_0 = u_1 = u_2 = 1$$

Par mémoïsation :

```python
mem = {0:1, 1:1, 2:1}


def u(n):
    if n not in mem:
        mem[n] = u(n-2) + u(n-3)
    return mem[n]
```

```python
def dyn_nb(n):
    lst = [1]

    for j in range(1, n+1):
        options = []
        for i in range(j):
            if direct_possible(i, j):
                options.append(lst[i]+1)
        lst.append(min(options))

    return lst
```

Minmax

```python
def minmax(etat, tourJ1):
    if [joueur1 a gagné]:
        return 1

    if [joueur2 a gagné]:
        return -1

    L = [coups possibles]

    if tourJ1:
        return max(L)
    else:
        return min(L)
```