TD #1 — Révisions (avec solutions)

Exercice 1. *Les œufs mystérieux (un grand classique)*

On vous donne un panier d'œufs mystérieux et vous indique une tour de N étages. Votre tâche est de déterminer à partir de quel étage lâcher un œuf le cassera. Vous savez que cet étage existe et est le même pour tous les œufs, que si un œuf n'est pas cassé (respectivement, est cassé) à un étage il ne sera pas cassé (resp., sera cassé) à tout étage inférieur (resp., supérieur). De plus, un œuf lâché mais non cassé n'est pas endommagé : il ne cassera pas plus bas qu'initialement.
L'objectif de cet exercice est de concevoir trois algorithmes permettant de déterminer le premier étage à partir duquel les œufs cassent, avec trois limites différentes sur le nombre *maximum* (c'est à dire, dans le pire cas) d'œufs que vous pouvez être amené à casser afin de trouver le résultat.
On demande seulement une description informelle (mais néanmoins précise !) des algorithmes.

1. Donnez un algorithme cassant *un* œuf dans le pire cas. Quel est le nombre maximum de lâchés effectués dans le pire cas ?

   On commence par lâcher l'œuf à l'étage 1, et tant qu'il n'est pas cassé on le lâche à l'étage immédiatement supérieur. On renvoie le numéro de l'étage où l'œuf casse. Le nombre maximum de lâchés effectués dans le pire cas est N (ou $N-1$ si l'on exploite la garantie d'existence d'un étage où les œufs cassent).

2. Donnez un algorithme effectuant le plus petit nombre de lâchés dans le pire cas. Quel est le nombre maximum d'œufs cassés dans le pire cas (à une constante près) ?

   On procède par dichotomie : on définit un intervalle d'étages candidats `b, t` valant initialement `1, N` et on lâche un œuf au milieu de l'intervalle `m = b + (t-b)//2`. S'il casse on met à jour l'intervalle à `b, m - 1`, et sinon à `m + 1, t`, et on continue récursivement jusqu'à atteindre un intervalle de taille 1. On renvoie l'étage ainsi déterminé si l'œuf s'y casse, ou celui immédiatement supérieur sinon. Le nombre maximum d'œufs cassés est égal au nombre maximum de lâchés, qui est un $O(\log(N))$, puisque la taille de l'intervalle candidat est (environ) divisée par deux à chaque étape.

3. Donnez un algorithme cassant *deux* œuf dans le pire cas, dont le nombre de lâchés effectués dans le pire cas est un $O(\sqrt{N})$.

   On définit $s = \lceil \sqrt{N} \rceil$, et on lâche un premier œuf aux étages de numéros $s, 2s, 3s, \ldots$ jusqu'à ce qu'il casse. Ceci nous prend au plus s lâchés puisque $s^2 \geq N$. On note t cet étage et a l'entier $t/s - 1$. On lâche maintenant un second œuf à partir de l'étage $as + 1$, et tant qu'il n'est pas cassé on le lâche à l'étage immédiatement supérieur. Ceci nous prend au plus s lâchés, et on renvoie le numéro de l'étage où l'œuf casse. Le nombre de lâchés dans le pire cas est majoré par $2s$, qui est bien un $O(\sqrt{N})$.

Exercice 2. *Graphes (adapté d'un roman de Georges Perec)*

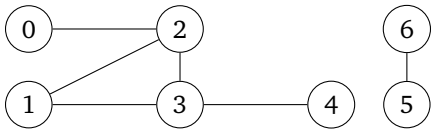
Ces échanges, qui sont suscités aussi bien par des occasions propices de vente ou d'achat (il s'agit alors de faire de la place) que par des inspirations subites, des lubies, des caprices ou des dégoûts, ne se font pas au hasard, et n'épuisent pas les douze possibilités de permutations qui pourraient se faire entre ces quatre lieux et que la figure 1 *met bien en évidence ; ils obéissent strictement au schéma de la* figure 2 *: quand Madame Marcia achète quelque chose, elle le met chez elle, dans son appartement, ou dans sa cave ; de là, ledit objet peut passer dans l'arrière-boutique, et de l'arrière-boutique dans le magasin ; du magasin enfin il peut revenir — ou parvenir s'il venait de la cave — dans l'appartement. Ce qui est exclu, c'est qu'un objet revienne dans la cave, ou arrive au magasin sans être passé par l'arrière-boutique, ou repasse du magasin dans l'arrière-boutique, ou de l'arrière-boutique dans l'appartement, ou enfin passe directement de la cave à l'appartement.*

1. Dessinez les graphes des figures 1 & 2.
2. Comment appelle-t'on les graphes semblables à celui de la figure 1.

Le graphe de la figure 1 est un graphe *complet* (ici orienté).

Exercice 3. *Graphes (adapté d'un exercice de Marie Durand)*

On donne le graphe non orienté suivant :



1. Donnez le degré du sommet 2.

   Le sommet 2 a un degré de 3 : il possède trois arêtes le reliant aux sommets 0, 1 et 3.

2. Ce graphe est-il acyclique ? Si non, donnez un exemple de cycle.

   Ce graphe n'est pas acyclique : il possède un cycle $2 \to 1 \to 3 \to 2$.

3. Ce graphe est-il connexe ? Si non, donnez les ensembles de sommets formant ses composantes connexes.

   Ce graphe n'est pas connexe : il est par exemple impossible d'atteindre le sommet 6 à partir du sommet 0. Il possède deux composantes connexes : $\{0, 1, 2, 3, 4\}$ et $\{5, 6\}$.

4. Donnez deux chemins allant du sommet 0 au sommet 4. Combien y a-t'il de chemins possibles entre ces sommets ?

   Un premier chemin est $0 \to 2 \to 3 \to 4$, et un second $0 \to 2 \to 1 \to 3 \to 4$. Il existe une infinité de chemins entre ces deux sommets car on peut notamment répéter le cycle $2 \to 1 \to 3 \to 2$ un nombre arbitraire de fois.

5. Donnez une représentation en Python de ce graphe par liste de listes d'adjacence, utilisant des `list`.

   On peut prendre la convention que le graphe sera représenté par une liste de listes, où chaque sommet est représenté par l'entier Python lui correspondant : soit `G` cette liste, `G[i]` donne la liste des sommets reliés par une arête au sommet représenté par `i`. On a alors par exemple :

   ```
   G = [[2], [2, 3], [1, 0, 3], [1, 2, 4], [3], [6], [5]]
   ```

   Cette représentation n'est pas unique : l'ordre d'énumération des sommets dans les sous-listes n'est pas fixé. On aurait aussi pu représenter le graphe avec des `tuple` à condition qu'il ne soit pas nécessaire de pouvoir le modifier.

6. De quelle autre façon pourrait-on représenter ce graphe ?

   Par exemple avec une matrice d'adjacence.

7. Écrivez une fonction Python `reachable(G, src, dst)` qui prend en entrée un graphe et les identifiants de deux sommets (représentés comme à la question précédente) et renvoie `True` (resp. `False`) s'il existe un chemin dans le graphe entre les deux sommets.

   Pour répondre à cette question, il suffit d'effectuer un parcours de graphe quelconque depuis `src` et de renvoyer `True` si `dst` est rencontré en chemin, et `False` si l'on a visité tous les sommets accessibles depuis `src` sans le rencontrer. On propose de faire cela avec un parcours en profondeur, mais ce n'est pas le seul choix possible. On donne ici une implémentation itérative utilisant un dictionnaire pour stocker les sommets visités, mais encore une fois ce n'est pas la seule option.

   ```
   def reachable(G, src, dst):
       vis = {src: True} # on stocke les sommets visités dans un dictionnaire
       tbe = G[src].copy() # on initialise une pile des sommets à explorer
                           # une copie est nécessaire pour ne pas modifier G
       while tbe: # while len(tbe) > 0:
           nxt = tbe.pop()
           if nxt == dst:
               return True
           elif nxt not in vis:
   ```

```
            vis[nxt] = True
            for s in G[nxt]:
                if s not in vis: # optionnel
                    tbe.append(s)
    return False
```

8. Donnez un plus-court-chemin entre les sommets 2 et 3. Quel algorithme pourrait être utilisé pour calculer un ensemble de plus-court-chemins entre le sommet 2 et tous les sommets qui lui sont accessibles ?

L'unique plus-court-chemin est ici $2 \to 3 \to 4$. Un algorithme de recherche en profondeur (d'origine 2) permet de répondre au problème posé. L'algorithme « de Dijkstra » permettrait également de répondre au problème, mais est ici inutilement compliqué et coûteux car le graphe est non pondéré.

9. Donnez une spécification adaptée à chacun des algorithmes suivants, qu'on suppose utilisés pour effectuer une recherche de plus-court-chemins :

i. Parcours en largeur.
ii. Algorithme « de Dijkstra ».
iii. Algorithme A^*

Tous ces algorithmes prennent en entrée un graphe (orienté ou non) représenté par liste de listes d'adjacence et un sommet source depuis lequel les plus-court-chemins seront calculés : ce sont des algorithmes de recherche de plus-court-chemin à origine unique (en anglais : *single source shortest path* ou *SSSP*). Leurs différences sont que :

i. Le parcours en largeur fournit les plus-court-chemins en terme de nombre d'arêtes ou d'arcs, c'est à dire les plus-court-chemins non pondérés.
ii. L'algorithme « de Dijkstra » prend en compte l'éventuelle pondération des arêtes ou des arcs. La version vue en ITC nécessite que cette pondération soit positive.
iii. L'algorithme A^* prend comme entrée supplémentaire un sommet destination, et vise uniquement à trouver un plus-court-chemin entre la source et la destination (ce n'est donc plus *vraiment* un SSSP). Il utilise pour cela une heuristique, et ne fournira une solution (garantie) correcte que si son heuristique est *admissible*.

Exercice 4. *Preuve d'algorithme*

On donne l'algorithme suivant :

```
1  # Renvoie True si l'entier positif x est présent dans
2  # la liste non vide triée (par ordre croissant) L d'entiers
3  # positifs et False sinon
4  def isIn(L:list, x):
5      bot = 0
6      top = len(L) - 1
7      while bot < top:
8          mid = bot + (top - bot)//2
9          if x == L[mid]:
10             return True
11         elif x < L[mid]:
12             bot, top = bot, mid - 1
13         else:
14             bot, top = mid + 1, top
15     if bot > top:
16         return False
17     else
18         return L[bot] == x
```

1. Quel est le nom du principe utilisé dans cet algorithme ?

Il s'agit d'une recherche dichotomique.

2. Montrez que cet algorithme termine toujours, en utilisant un variant approprié.

Si la liste est de taille 1, la condition de l'unique boucle `while` est initialement fausse, et l'algorithme termine trivialement. Si la condition du test de la ligne 9 est vraie au cours de l'exécution, l'algorithme termine à la ligne 10. On suppose donc maintenant que la condition de la ligne 9 est toujours fausse et que `bot < top` (ce qui est vrai initialement pour toute liste de taille au moins 2) ; on va montrer que cette dernière condition deviendra toujours fausse à une certaine itération, ce qui permettra de conclure. Pour cela, on montre que le variant `top - bot` décroît strictement à chaque itération (une valeur nulle ou négative de cette quantité entraînant la fin de l'exécution). La paire `(bot, top)` étant modifiée ou bien en `(bot, mid - 1)` (ligne 12) ou bien en `(mid + 1, top)` (ligne 14), il suffit de montrer qu'on a toujours $0 \leq$ `bot` $\leq$ `mid` $\leq$ `top` après la ligne 8, ce qu'on fait par induction : la condition d'entrée de boucle et le fait que `bot` ≥ 0 (initialement vrai, puis par induction) font que `top - bot` est positif, et donc on a bien `bot` $\leq$ `mid` après la ligne 8 ; `mid` $\leq$ `top` est donné par le fait que `mid - bot` $=$ `(top - bot)//2` $\leq$ `top - bot`.

3. Montrez que cet algorithme renvoie toujours la bonne réponse, en utilisant un invariant.

Si la liste est de taille 1, l'algorithme renvoie la bonne réponse à la ligne 18. Si la condition du test de la ligne 9 est vraie au cours de l'exécution, l'algorithme renvoie la bonne réponse à la ligne 10. On doit donc montrer que l'algorithme renvoie la bonne réponse dans les deux autres cas : ligne 16 et ligne 18 pour une liste de taille supérieure à 1. Pour cela, on utilise l'invariant suivant : *si* `x` *est présent dans la liste, il ne peut l'être qu'à un indice* `i` $\in [\![$`bot, top`$]\!]$. Ceci permettra de conclure dans les deux cas : à la ligne 16, l'intervalle $[\![$`bot, top`$]\!]$ est vide donc `i` ne peut pas exister et la réponse `False` donnée par l'algorithme est correcte ; à la ligne 18 on a nécessairement `bot == top` et l'algorithme teste si `x` est présent à l'unique indice encore possible.
On montre maintenant l'invariant par induction. Il suffit d'observer que celui-ci est initialement vrai et que si la condition du test de la ligne 11 est vraie (resp. fausse), alors par le fait que L est triée `x` ne peut pas être présent à un indice supérieur (resp. inférieur) à `mid`, et par induction la mise à jour de `bot` et `top` effectuée à la ligne 12 (resp. 14) maintient l'invariant.

4. Quelle propriété de correction est démontrée par les deux questions précédentes ?

Ces questions montrent la *correction totale* de l'algorithme : il termine toujours et renvoie toujours une bonne réponse en cas de terminaison.

5. Donnez la complexité pire cas de cette fonction en prenant comme métrique le nombre d'accès à la liste L.

Chaque itération de l'unique boucle de l'algorithme effectue un nombre constant d'accès à L, et le nombre d'accès hors de la boucle est également constant. On a montré à une question précédente que la quantité `top - bot` décroît strictement à chaque itération, et on va maintenant montrer qu'à une constante additive près, chaque itération divise cette quantité par deux. On note `top'` et `bot'` les valeurs de `top` et `bot` après mise à jour à la ligne 12 ou 14. On a alors : ligne 12) `top' - bot'` $\approx$ `(top - bot)//2` ; ligne 14) `top' - bot'` $\approx$ `(top - bot) - (top - bot)//2` $\approx$ `(top - bot)//2`. Il s'ensuit que le nombre d'itération de la boucle est un $O(\log(N))$ avec N le nombre d'éléments de L. C'est aussi un $\Omega(\log(N))$ si la condition du test à la ligne 9 n'est jamais vraie, et le nombre d'accès effectués dans le pire cas est donc un $\Theta(\log(N))$.

N.B. On peut montrer que cette complexité est optimale pour les algorithmes qui ne peuvent apprendre des informations sur la liste qu'en comparant des éléments d'indices connus avec l'élément recherché.

Exercice 5. *Dictionnaires*

1. Donnez une définition succincte des dictionnaires Python.

Les dictionnaires Python servent à représenter et manipuler des associations *(clef, valeur)*. Une *clef* donnée ne peut être présente qu'une seule fois dans un même dictionnaire à équivalence près, et il n'y a pas de restriction particulière pour les valeurs.

2. Rappelez la (ou une) syntaxe permettant de :

i. créer un dictionnaire vide ;
ii. créer un dictionnaire avec initialisation (par exemple associant la valeur `True` à la clef `"martre des pins"`) ;
iii. accéder à la valeur associée à une clef ;
iv. associer une valeur à une clef (existante ou non) ;
v. tester si une clef est présente dans un dictionnaire ; n'est pas présente ;

vi. itérer sur les clefs d'un dictionnaire ;
vii. obtenir le nombre de couples stockés dans un dictionnaire.

i. `D = {}`
ii. `D = {"martre des pins" : True}`
iii. `D[k]` pour une clef `k`
iv. `D["martre des pins"] = True`
v. `"martre des pins" in D` ; `"martre des pins" not in D`
vi.
```
for k in D:
```
vii. `len(D)`

Soit N un entier naturel, on considère $\mathcal{I}_N$ l'ensemble des entiers naturels strictement inférieurs à N.

3. Combien $\mathcal{I}_N$ possède-t'il de sous-ensembles distincts ?

2^N.

On souhaite représenter et manipuler les sous-ensembles de $\mathcal{I}_N$, à travers cinq fonctions :

— `create(N)` qui crée et renvoie une représentation du sous-ensemble vide de $\mathcal{I}_N$;
— `test(E, i)` qui teste si l'entier `i` est présent dans le sous-ensemble `E` ;
— `add(E, i)` qui ajoute (si nécessaire) l'entier `i` au sous-ensemble `E` ;
— `rem(E, i)` qui supprime (si nécessaire) l'entier `i` du sous-ensemble `E` ;
— `size(E)` qui renvoie le cardinal du sous-ensemble `E`.

4. Proposez une solution à base de `list`.

On propose :
```
def create(N):
    return [False for i in range(N)]

def test(E, i):
    if i < 0 or i >= len(E):
        return False
    return E[i]

def add(E, i):
    if i < 0 or i >= len(E):
        return
    E[i] = True

def rem(E, i):
    if i < 0 or i >= len(E):
        return
    E[i] = False

def size(E):
    sz = 0
    for v in E:
        if v:
            sz = sz + 1
    return sz
```

5. Proposez une solution à base de `dict`. On pourra utiliser la construction (hors programme) `del D[k]` pour supprimer la clef `k` (et sa valeur associée) du dictionnaire `D` ; attention, la clef *doit* être présente dans le dictionnaire.

On propose :
```
def create(N):
    return {"max":N}

def test(E, i):
```

```
    return i in E

def add(E, i):
    if i < 0 or i >= E["max"]:
        return
    E[i] = True

def rem(E, i):
    if i in E:
        del E[i]

def size(E):
    return len(E) - 1
```

On considère maintenant un ensemble $\mathcal{S}$ implicite quelconque, dont on souhaite toujours représenter et manipuler les sous-ensembles comme précédemment (où `create` ne prend plus d'argument, et où l'on remplace simplement `i` par un élément `x` quelconque).

6. Proposez une solution à base de `list`.
Rappel : la fonction-méthode `pop` des `list` est uniquement au programme dans sa version *sans argument*. Vous *ne pouvez pas* utiliser « `L.pop(i)` ».

On propose :
```
def create():
    return []

def test(E, x):
    for v in E:
        if v == x:
            return True
    return False

def add(E, x):
    for v in E:
        if v == x:
            return
    E.append(x)

def rem(E, x):
    sz = len(E)
    for i in range(sz):
        if E[i] == x:
            E[i] = E[sz - 1]
            E.pop()
            return
    return

def size(E):
    return len(E)
```

7. Proposez une solution à base de `dict`.

On propose :
```
def create():
    return {}

def test(E, x):
    return x in E

def add(E, x):
    E[x] = True
```

```
def rem(E, x):
    if x in E:
        del E[x]

def size(E):
    return len(E)
```