Devoir surveillé d'informatique

Consignes

- La calculatrice n'est **pas autorisée**.
- On pourra toujours librement utiliser une fonction demandée à une question précédente même si cette question n'a pas été traitée.
- Veillez à présenter vos idées et vos réponses partielles même si vous ne trouvez pas la solution complète à une question.
- La clarté et la lisibilité de la rédaction et des programmes sont des éléments de notation.

❑ Exercice 1 : *Questions de cours*

1. Sur la récursivité
   a) Donner la définition d'une fonction récursive.

   Une fonction récursive est une fonction qui s'appelle elle-même.

   b) Quel serait le comportement d'une fonction récursive dépourvue de condition d'arrêt ?

   En l'absence de condition d'arrêt la fonction s'appelle elle-même à l'infini et donc ne s'arrête jamais.

   c) Ecrire une fonction *iterative* `occurrences` qui prend en argument une chaine de caractères `s` et un caractère `c` et qui renvoie le nombre d'apparitions du caractère `c` dans la chaine `s`. Par exemples, `occurrences("un exemple",'e')` renvoie `3` et `occurrences("un exemple",'a')` renvoie `0`.

```python
def occurrences(s, c):
    '''Renvoie le nombre d'apparition de c dans s'''
    cpt = 0
    for l in s:
        if l == c:
            cpt += 1
    return cpt
```

   d) Ecrire une version *récursive* de cette fonction.
   🛟 Indication : on rappelle que si `s` une chaine de caractères alors `s[1:]` est la chaine `s` privée de son premier caractère. Par exemple `"Python"[1:]` est la chaine `"ython"`.

```python
def occurrences_rec(s, c):
    if s == "":
        return 0
    else:
        if s[0] == c:
            return 1 + occurrences_rec(s[1:], c)
        else:
            return occurrences_rec(s[1:], c)
```

2. Sur le tri par sélection
   a) Expliquer brièvement le principe de l'algorithme du *tri par sélection* et donner l'évolution du contenu de la liste `[42, 28, 11, 9, 33]` lorsqu'elle est triée en utilisant cet algorithme.

On parcourt la liste avec un indice `i` à partir du début. Pour chaque valeur de `i` on sélectionne l'indice `m` du minimum des éléments de la liste *à partir* de l'indice `i` et on l'échange avec celui d'indice `i`. Sur la liste `[42, 28, 11, 9, 33]`, on obtient successivement :

- `[9, 28, 11, 42, 33]`
- `[9, 11, 28, 42, 33]`
- `[9, 11, 28, 42, 33]`
- `[9, 11, 28, 33, 42]`

b) Ecrire en Python une fonction `echange` qui prend en argument une liste `lst` ainsi que deux entiers `i` et `j` ne renvoie rien et échange les éléments d'indice `i` et `j` de cette liste. Par exemple, si `lst = [25, 17, 34, 22, 37]` alors `echange(lst, 1, 3)` doit modifier la liste `lst` en `[25, 22, 34, 17, 37]`. On vérifiera les préconditions sur `i` et `j` à l'aide d'instructions `assert`.

```
def echange(lst, i, j):
    '''Echange les éléments d'indice i et j dans lst'''
    assert 0<= i < len(lst) and 0<=j < len(lst)
    lst[i], lst[j] = lst[j], lst[i]
```

c) Ecrire en Python une fonction `indice_min_depuis` qui prend en argument une liste `lst` et un entier `i` et renvoie l'indice du plus petit élément de la liste `lst` à partir de l'indice `i`. Par exemple, si `lst = [25, 17, 34, 22, 37]` alors `indice_min_depuis(lst, 2)` doit renvoyer `3` (c'est à dire l'indice de 22 qui est l'élément minimal à partir de celui d'indice 2).

```
def indice_min_depuis(lst, i):
    '''Renvoie l'indice du minimum de lst à partir de l'indice i'''
    imin = i
    for j in range(i+1, len(lst)):
        if lst[j] < lst[imin]:
            imin = j
    return imin
```

d) Ecrire en Python une fonction `tri_selection` qui prend en argument une liste `lst` et modifie cette liste de façon à la trier par ordre croissant en utilisant l'algorithme du tri par sélection. On utilisera les fonctions `echange` et `indice_min_depuis` écrites aux questions précédentes.

```
def tri_selection(liste):
    '''Trie la liste  par sélection'''
    for i in range(len(liste)):
        imin = indice_min_depuis(liste, i)
        echange(liste, i, imin)
```

e) Un ordinateur a pris 3 secondes pour trier une liste de 250 000 éléments en utilisant un tri par sélection. Donner une estimation (aucune justification n'est demandée) du temps que prendrait cet ordinateur pour trier une liste de 1 000 000 éléments en utilisant le même algorithme.

On peut estimer que le temps de calcul est proportionnel au carré du nombre d'éléments à trier. Ainsi, si le temps est de 3 secondes pour 250 000 éléments, il serait de $3 \times \left(\frac{1000000}{250000}\right)^2 = 3 \times 16 = 48$ secondes pour 1 000 000 éléments. La justification n'était pas demandée mais en examinant l'algorithme on peut constater que le nombre d'opérations à effectuer évolue comme le carré du nombre d'éléments à trier.

3. Sur la représentation binaire des entiers

a) Donner l'écriture en base 2 de $(172)_{10}$.

> $(172)_{10} = (10101100)_2$

b) Ecrire $(10011101)_2$ en base 10.

> C'est $(10011101)_2 = 1\times 2^7 + 0\times 2^6 + 0\times 2^5 + 1\times 2^4 + 1\times 2^3 + 1\times 2^2 + 0\times 2^1 + 1\times 2^0 = 128 + 16 + 8 + 4 + 1 = 157$

c) Quel est le plus grand nombre représentable en base 2 en utilisant 10 bits ?

> Le plus grand nombre représentable en base 2 avec 10 bits est $(1111111111)_2 = 2^{10} - 1 = 1023$

d) On suppose que l'on travaille sur des entiers codés sur 8 bits en complément à 2. A quel nombre entier en base 10 correspond $(11011001)_2$?

> Le bit de poids fort est compté négativement et les autres positivement. Donc $(11011001)_2 = -2^7 + 2^6 + 2^4 + 2^3 + 2^0 = -128 + 64 + 16 + 8 + 1 = -39$

❑ Exercice 2 : *Permutation des éléments d'une liste*

1. Le but de cette partie est de dénérer *une* permutation aléatoire des éléments d'une liste
Le *mélange de Fischer-Yates* (aussi appelé *mélange de Knuth*) est un algorithme de mélange aléatoire des éléments d'une liste. Il consiste à parcourir la liste en partant de la fin avec un indice `i` et à échanger l'élément d'indice `i` avec un élément choisi aléatoirement parmi les éléments ayant un indice inférieur *ou égal*. Voici un exemple du déroulement de l'algorithme (crédit : Wikipedia) sur la liste `['E','L','V','I','S]` on part de la fin (donc de l'indice 4) :

Etat de la liste	Indice `i`	Choix aléatoire dans $[\![0;i]\!]$	Liste après échange
`['E','L','V','I','S']`	4	4	`['E','L','V','I','S']`
`['E','L','V','I','S']`	3	0	`['I','L','V','E','S']`
`['I','L','V','E','S']`	2	2	`['I','L','V','E','S']`
`['I','L','V','E','S']`	1	0	`['L','I','V','E','S']`

a) Donner le déroulement de l'algorithme sur la liste `['A','B','C','D','E','F']` en supposant que les choix aléatoires successifs sont : `2, 0, 3, 1, 0`.

> On obtient les évolutions suivantes de la liste :
> - `['A','B','F','D','E','C']` (élément d'indice 5 :`'F'` échangé avec celui d'indice 2 : `'C'`)
> - `['E','B','F','D','A','C']` (élément d'indice 4 échangé avec celui d'indice 0)
> - `['E','B','F','D','A','C']` (élément d'indice 3 échangé avec celui d'indice 3)
> - `['E','F','B','D','A','C']` (élément d'indice 2 échangé avec celui d'indice 1)
> - `['F','E','B','D','A','C']` (élément d'indice 1 échangé avec celui d'indice 0)

b) Afin de générer une entier aléatoire on veut utiliser la fonction `randint` du module `math`. Que faut-il écrire en début de programme pour que cette fonction soit utilisable ?

```
1 from random import randint
```

c) Ecrire en Python une fonction *itérative*, `melange_iteratif` qui prend en argument une liste `lst` ne renvoie rien et mélange aléatoirement les éléments de la liste en utilisant l'algorithme de Fischer-Yates. On supposera déjà écrite une fonction `echange` qui prend en argument une liste ainsi que deux indices et échange les éléments situés à ces indices.

```python
def melange_iteratif(lst):
    for i in range(len(lst)-1, 0, -1):
        j = randint(0, i)
        echange(lst, i, j)
```

d) Ecrire en Python une fonction *récursive*, `melange_recursif` qui prend en argument une liste `lst` ne renvoie rien et mélange aléatoirement les éléments de la liste en utilisant l'algorithme de Fischer-Yates.
🛟 Indication : on pourra fournir à cette fonction un paramètre entier `i` indiquant l'indice de l'élément qui sera potentiellement échangé avec l'un de ceux qui le précède.

```python
def melange_recursif(lst, i):
    if i> 0:
        j = randint(0, i)
        echange(lst, i, j)
        melange_recursif(lst, i-1)
```

2. Le but de cette partie est de générer la liste de *toutes* les permutations d'une liste.
On souhaite écrire une fonction `permutation` qui prend en argument une liste et renvoie la liste de toutes les permutations de cette liste. Par exemple `permutation(['I','T','C'])` doit renvoyer la liste : `[['I','T','C'], ['I','C','T'], ['T','I','C'], ['T','C','I'], ['C','T','I'], ['C','I','T']]`.
Pour cela on propose l'algorithme *récursif* suivant : on commence par générer la permutation de la sous liste commençant à l'indice 1 puis on insère le premier élément à toutes les positions possibles dans les listes obtenues. Sur l'exemple de la liste `['I','T','C']`, on génère donc les permutations de `['T','C']` c'est à dire `[['T','C'], ['C','T']]`. Puis on insère `'I'` à toutes les positions possibles dans ces listes ce qui donne :
`[['I','T','C'], ['T','I','C'], ['T','C','I'], ['I','C','T'], ['C','I','T'], ['C','T','I']]`.

a) Ecrire la fonction `insere` qui prend en argument un élément `elt` une liste `lst` et renvoie la liste de listes obtenues en insérant `elt` à toutes les positions possibles dans `lst`. Par exemple, si `lst = ['T','C']` et `elt = 'I'` alors `insere(elt, lst)` doit renvoyer `[['I','T','C'], ['T','I','C'], ['T','C','I']]`.

```python
def insere(lst, elt):
    return [lst[:i] + [elt] + lst[i:] for i in range(len(lst)+1)]
```

b) Ecrire en Python une fonction *récursive*, `permutation` qui prend en argument une liste `lst` et renvoie la liste des permutations de la liste `lst`.
🛟 Indication : on pourra générer *récursivement* les permutations de `lst` privé de son premier élément puis insérer cet élément à tous les emplacements possibles grâce à la fonction précédente.

```python
def permutation(lst):
    if len(lst) == 0:
        return [[]]
    temp =  permutation(lst[1:])
    res = []
    for x in temp:
        res += insere(x, lst[0])
    return res
```

❑ **Exercice 3** : *Problème du sac à dos*
On dispose d'un sac à dos pouvant contenir un poids maximal noté P et de n objets ayant chacun un poids $(p_i)_{0\leqslant i\leqslant n-1}$ et une valeur $(v_i)_{0\leqslant i\leqslant n-1}$. On cherche à remplir le sac à dos de manière à maximiser la valeur totale des objets contenus dans le sac sans dépasser le poids maximal P. Par exemple, si on dispose des objets suivants :

— un objet de poids $p_0 = 4$ et de valeur $v_0 = 20$,

— un objet de poids $p_1 = 5$ et de valeur $v_1 = 28$,

— un objet de poids $p_2 = 6$ et de valeur $v_2 = 36$,

— un objet de poids $p_3 = 7$ et de valeur $v_3 = 50$,

et qu'on suppose que le poids maximal du sac est 10 alors un choix possible serait de prendre l'objet 3, aucun autre objet ne rentre alors dans le sac et la valeur du sac est de 50 avec un poids de 7. Une autre possibilité plus intéressante serait de choisir les objets 0 et 2, la valeur totale serait alors de 56 et le poids du sac de 10.

Dans toute la suite de l'exercice on supposera que les poids et les valeurs des objets sont fournis sous la forme d'une liste de Python contenant les tuples `(poids, valeur)` représentant les objets. Par exemple, les objets précédents seraient représentés par la liste suivante :

```
objets = [(4, 20), (5, 28), (6, 36), (7, 50)]
```

1. On considère la stratégie gloutonne suivante : on trie les objets par ordre décroissant de leur rapport valeur/poids et on les prend dans cet ordre jusqu'à ce que le poids maximal soit atteint.

   a) Vérifier qu'en appliquant cette stratégie à la liste d'objets :

   ```
   [(4, 30), (5, 34), (6, 36), (7, 49), (10, 74)]
   ```

   et un poids maximal de 10, on n'obtient pas la meilleure solution.

   > En triant les objets par rapport poids/valeur, on obtient l'ordre suivant :
   > — objet 0 (poids : 4, valeur : 30, rapport : 7.5)
   > — objet 4 (poids : 10, valeur : 74, rapport : 7.4)
   > — objet 3 (poids : 7, valeur : 49, rapport : 7.0)
   > — objet 1 (poids : 5, valeur : 34, rapport : 6.8)
   > — objet 2 (poids : 6, valeur : 36, rapport : 6.0)
   > On parcourt donc cette liste en prenant les objets tant qu'ils ne dépassent pas la contrainte de poids. Cela conduit à choisir les objets 0 et 1 pour une valeur de 64. Cette combinaison n'est pas optimale puisque l'on peut obtenir une valeur de 74 en prenant uniquement l'objet 4.

   b) Ecrire une fonction `glouton`, qui prend en arguments une liste d'objets qu'on suppose *déjà triée par ordre décroissant du rapport valeur/poids* et un poids maximal et qui renvoie la valeur maximale que l'on peut obtenir en appliquant la stratégie gloutonne.

   ```python
   def glouton(objets, pmax):
       poids = 0
       valeur = 0
       for obj in objets:
           p, v = obj
           if poids + p <= pmax:
               poids += p
               valeur = valeur + v
       return valeur
   ```

   c) Montrer sur un exemple de votre choix (avec au moins 3 objets) que si on choisit de trier les objets par ordre croissant de poids (c'est à dire qu'on choisit en premier les objets les plus légers), alors la stratégie gloutonne ne donne pas non plus la solution optimale.

   > On peut par exemple considérer les objets suivants : `[(1, 10), (3, 20), (5, 40)]` et un poids maximal de 5. La stratégie gloutonne choisirait les objets 0 et 1 pour une valeur de 30 alors que la solution optimale est de prendre l'objet 2 pour une valeur de 40.

2. La recherche exhaustive consiste à énumérer tous les choix possibles d'objet et à calculer la valeur ainsi que le poids pour chaque choix, on retient alors le choix qui maximise la valeur du sac sans dépasser le poids maximal. On propose de représenter un choix d'objets par une liste contenant des 0 et des 1. Si le i-ème élément de la liste vaut 1 alors l'objet i est choisi, s'il vaut 0 alors l'objet i n'est pas choisi. Par exemple, pour les objets précédents, le choix de prendre uniquement l'objet 3 serait représenté par la liste `[0, 0, 0, 1]` et le choix de prendre les objets 0 et 2 serait représenté par la liste `[1, 0, 1, 0]`.
   a) Justifier rapidement que le nombre possible de choix d'objet est 2^n.

   > Pour chaque objet, on a deux choix possibles, le prendre ou non, comme il y a n objets, le nombre total de choix est 2^n.

   b) Afin de générer tous les choix possibles on propose de parcourir les entiers de 0 à 2^n-1 et de convertir chaque entier en son écriture binaire. Ecrire une fonction `choix` qui prend en argument un entier `n` et un entier `k` tel que $0 \leqslant k \leqslant 2^n - 1$ et renvoie la liste des chiffres de k en base 2 sur n bits. Par exemples `choix(8,17) renvoie [0,0,0,1,0,0,0,1]` (car on utilise `8` bits et $(00010001)_2 = 17$), et `choix(8,121) renvoie [0,1, 1, 1, 1, 0, 0, 1]`
   🛟 Indication : on pourra utiliser la fonction `bin` de Python qui prend en argument un nombre entier et renvoie son écriture en base 2 précédée de `0b` sous la forme d'une chaine de caractère, par exemple `bin(17)` renvoie `"0b10001"`.

```python
def choix(n,k):
    # Chaine de caractères contenant les bits de k
    bits = bin(k)[2:]
    # On rajoute des 0 à gauche pour obtenir une chaine de longueur n
    bits = '0'*(n-len(bits)) + bits
    # on transforme en liste de 0 et de 1:
    return [int(b) for b in bits]
```

   c) Ecrire une fonction `valeur_poids` qui prend en arguments, une liste d'objets ainsi qu'un choix d'objet (sous la forme indiquée ci-dessus) et qui renvoie le poids et la valeur du sac correspondant à ce choix. Par exemple avec la liste `objets` donnée en exemple plus haut, `valeur_poids(objets, [1, 0, 1, 0])` doit renvoyer $(56, 10)$.

```python
def valeur_poids(objets, choix):
    v = 0
    p = 0
    for i in range(len(choix)):
        if choix[i] == 1:
            pobj, vobj = objets[i]
            v = v + vobj
            p = p + pobj
    return v, p
```

   d) Ecrire une fonction `recherche_exhaustive` qui prend en argument une liste d'objets et un poids maximal et qui renvoie la valeur maximale que l'on peut obtenir en appliquant la stratégie de recherche exhaustive. C'est à dire en testant tous les choix possibles d'objets et en retenant le choix de valeur maximale qui ne dépasse pas le poids maximal autorisé.

```python
def recherche_exhaustive(objets, pmax):
    n = len(objets)
    meilleure_valeur = 0
    for k in range(2**n):
        choix = choix(n, k)
        v, p = valeur_poids(objets, choix)
        if p <= pmax and v > meilleure_valeur:
            meilleure_valeur = v
    return meilleure_valeur
```