

DS d'informatique n°2

(21/01/26)

Consignes :

- La correction tient compte du soin apporté à la présentation et à la rédaction de la copie. Les bouts de code devront être écrits en langage Python en respectant la syntaxe inhérente à ce langage. On veillera notamment à faire apparaître clairement les indentations qui structurent le code.
- On pourra utiliser une (ou plusieurs) fonction(s) définie(s) à une question précédente, même si celle-ci n'a pas été résolue.
- Sauf mention contraire, les fonctions à écrire ne doivent pas modifier leurs entrées.
- Lorsque l'énoncé impose des contraintes sur le type et la valeur des arguments d'une fonction, on supposera que ces contraintes sont vérifiées lors de l'appel de la fonction. Il n'est pas nécessaire d'ajouter des tests dans la fonction pour vérifier ces contraintes.

Exemple : Si l'énoncé demande de définir une fonction d'entête $f(x)$ avec x un nombre réel positif, on supposera que lors de l'appel x est un nombre réel positif. Il n'est pas nécessaire d'ajouter des tests dans la fonction f pour vérifier si x est de type `float` et, si oui, si il est bien positif.

- L'usage des calculatrices et de tout document annexe n'est pas autorisé.
- Les seules fonctions natives de Python autorisées dans ce devoir sont celles rappelées dans le tableau ci-dessous. En particulier, ni les fonctions `min()`, `max()`, `sum()`, `count()`, ni les méthodes de tris de type `sorted()`, `sort()` ne sont autorisées.
- Le barème de chaque question contient des points intermédiaires. Une réponse incomplète peut rapporter des points.

Rappels d'éléments du langage Python :

Dans le tableau ci-dessous, L représentent une liste Python.

Fonction	But
<code>len(L)</code>	Renvoie la longueur (nombre d'éléments) de la liste L
<code>L.append(e)</code>	Modifie L en lui ajoutant l'élément e en dernière position

Présentation

Lara C., célèbre exploratrice/pilleuse de tombes, vient de récupérer le plan d'une tour ancienne. Cette tour a été conçue de la manière suivante : une pièce unique occupe l'intégralité du rez-de-chaussée de la tour; le nombre de pièces augmente ensuite de 1 à chaque étage (2 pièces au 1er étage, 3 pièces au 2ème étage, etc.). Elle a aussi pour particularité de n'autoriser que des déplacements verticaux. À partir d'une pièce donnée, on ne peut accéder qu'à deux pièces situées à l'étage supérieur. Il est impossible d'accéder directement aux autres pièces situées au même étage.

Chaque pièce contient une certaine quantité de butin mais le temps pour le récupérer est limité. De vrais archéologues arriveront bientôt pour sécuriser le site et en limiter l'accès. L'objectif est donc de trouver le chemin optimal dans la tour pour récupérer le maximum de butin en faisant un unique trajet du bas vers le haut de la tour, sans jamais redescendre.

Exemple : La **figure 1** représente un exemple d'une tour de $n = 4$ étages. Le chemin optimal dans cette tour est indiqué par les cases colorées. Il permet de récupérer un butin total égal à $2 + 11 + 2 + 12 = 27$ k€.

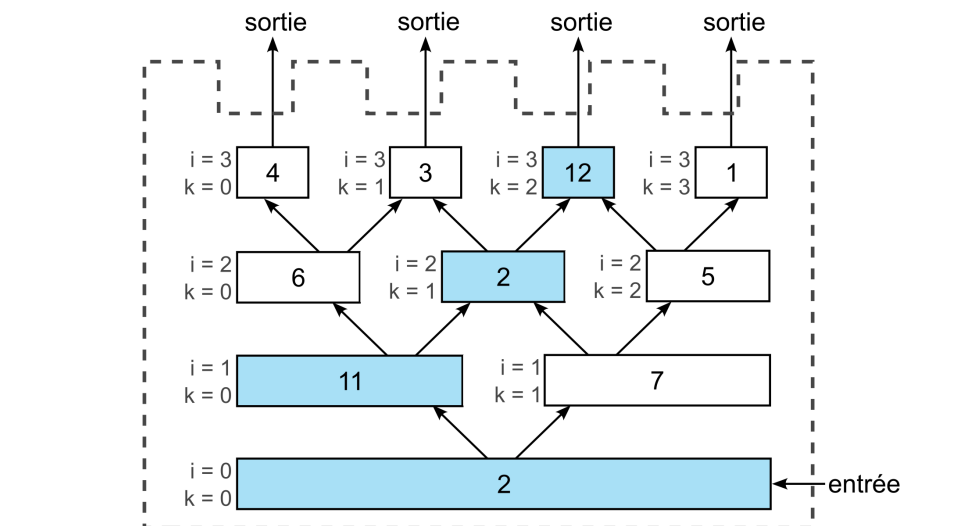


Figure n°1

Notation et implémentation en Python

Dans la suite de l'énoncé, on adoptera les notations suivantes :

- n est le nombre d'étages de la tour (incluant le rez-de-chaussée).
- i est l'indice d'un étage particulier. Le rez-de-chaussée est l'étage d'indice $i = 0$. Le dernier étage de la tour a pour indice $i = n - 1$
- k représente la position d'une pièce au sein d'un étage. Il y a $(i + 1)$ pièces différentes à l'étage d'indice i . La pièce la plus à gauche d'un étage a toujours pour indice $k = 0$. La pièce la plus à droite a donc pour indice $k = i$.
- La quantité de butin dans chaque pièce est stockée dans une liste de listes Python :
 - La sous-liste d'indice i contient la liste des butins de toutes les pièces de l'étage d'indice i de la tour.
 - L'élément d'indice k de la sous-liste d'indice i contient le butin de la pièce d'indice k à l'étage d'indice i de la tour.

Exemple : La tour représentée sur la **figure 1** serait implémentée en Python par la liste suivante :

```
[[2], [11, 7], [6, 2, 5], [4, 3, 12, 1]]
```



Tous les algorithmes écrits dans la suite devront être capable de s'adapter à des listes de tailles variables. Il n'y aura pas toujours 4 étages dans la tour comme dans l'exemple précédent.

I. Préparation de l'expédition

I.1. Analyse rapide du plan de la tour

On suppose qu'une tour a été implémentée en Python en définissant une liste nommée `tour_ex`. Cette tour contient plus de 10 étages.

1. Écrire une séquences d'instruction en langage Python permettant de définir :
 - une variable `n` contenant le nombre exact d'étages de la tour;
 - une variable `butins_etage` contenant la liste des butins de l'étage d'indice 5 de la tour;
 - une variable `butin_piece` contenant le butin de la deuxième pièce en partant de la gauche à l'étage d'indice 3 de la tour.

On suppose qu'un chemin optimal dans une tour a été identifié. Le **butin** de chaque pièce traversée est mémorisé dans une liste `chemin`. On souhaite calculer le butin total correspondant à ce chemin.

2. Définir une fonction Python d'en-tête `calcul_somme(chemin)` qui :
 - prend en argument une liste de nombres `chemin`;
 - renvoie la somme des éléments de la liste `chemin`.

Exemple : L'appel `calcul_somme([3, 7, 1])` doit renvoyer l'entier 11.

I.2. Création du plan d'une tour

On suppose qu'une fonction d'en-tête `alea(min,max)` a été définie dans un code Python. Cette fonction prend en arguments deux entiers `min` et `max` (avec `min < max`) et renvoie un entier choisi aléatoirement dans l'intervalle `[[min;max]]`.

Exemple : L'appel `alea(3,17)` a renvoyé l'entier 8.

3. Définir une fonction Python d'en-tête `creer_etage(nb_pieces)` qui :
 - prend en argument un entier strictement positif `nb_pieces`;
 - renvoie une liste de `nb_pieces` entiers choisis aléatoirement entre 1 et 20 inclus et correspondant à la liste des butins dans un étage donné de la tour.

Exemple : L'appel `creer_etage(5)` a renvoyé la liste `[14, 12, 19, 9, 3]`.

4. Définir une fonction Python d'en-tête `creer_tour(n)` qui :
 - prend en argument un entier strictement positif `n` correspondant au nombre d'étages de la tour;
 - renvoie une liste de listes correspondant à l'implémentation d'une tour de `n` étages, qui respecte le format décrit dans la partie « Présentation » en début d'énoncé.

La fonction `creer_tour` **doit** faire appel à la fonction `creer_etage` de la question précédente.

Exemple : L'appel `creer_tour(3)` a renvoyé la liste `[[9], [12, 4], [3, 11, 10]]`.

II. Recherche du chemin optimal

Il existe 2^{n-1} chemins différents dans une tour de n étages. Pour des valeurs de n élevées, étudier une par une toutes les possibilités prendrait beaucoup trop de temps. On étudie dans la suite deux stratégies différentes pour rechercher de manière plus efficace le chemin qui maximise le butin récupéré.

II.1. Méthode n°1 : stratégie gloutonne

Dans une méthode dite « gloutonne », un choix optimal local est réalisé à chaque étape. Cette méthode n'assure pas de conduire à la solution globalement optimale. Mais elle est rapide et on peut espérer obtenir une solution pas trop mauvaise.

La stratégie gloutonne est la suivante : quand Lara est dans une pièce située aux coordonnées (i, k) dans la tour et qu'elle doit choisir un des **deux** escaliers menant à l'étage supérieur, elle choisit toujours celui qui conduit à la pièce ayant le butin le plus élevé. On rappelle qu'il n'y a **que deux pièces accessibles** depuis une pièce qui n'est pas située au dernier étage de la tour.

5. Représenter le chemin obtenu en appliquant la stratégie gloutonne à la tour représentée sur la **figure 2** ci-dessous. On pourra hachurer sur le document-réponse les pièces qui ont été explorées par Lara. Pour cet exemple, la solution gloutonne est-elle la solution optimale? Justifier.

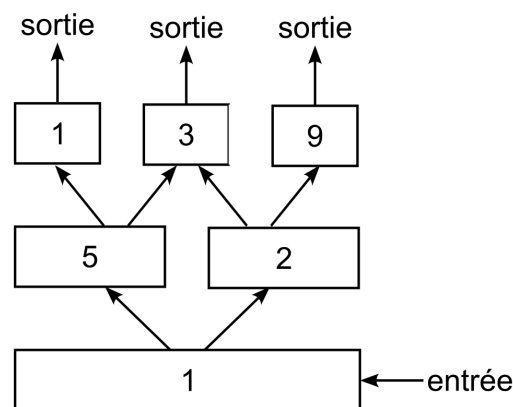


Figure n°2

6. On suppose que Lara est dans la pièce de coordonnées (i, k) . Quelles sont les coordonnées des deux salles accessibles depuis cette pièce?
7. Définir une fonction Python d'en-tête `methode_1(tour)` qui :
- prend en argument une liste de listes `tour` contenant le butin dans chaque pièce de la tour (liste décrite dans la partie « Présentation » en début d'énoncé);
 - renvoie une liste `chemin` contenant la liste des butins des pièces explorées en utilisant la stratégie gloutonne. Le premier élément de `chemin` est le butin de la pièce située au rez-de-chaussée. Le dernier élément de `chemin` est le butin de la dernière pièce avant la sortie. Le nombre final d'élément de `chemin` est donc égal au nombre d'étages de la tour.

Il peut être utile de mémoriser la position k à chaque étage pour savoir comment progresser dans la tour.

Exemple : L'appel `methode_1([[6], [16, 6], [7, 18, 20], [5, 17, 16, 15]])` a renvoyé la liste `[6, 16, 18, 17]`.

II.2. Méthode n°2 : recherche récursive du chemin optimal

On souhaite écrire une fonction **récursive** `calcul_sommes_max_rec` qui prend en arguments :

- une liste de listes `tour` dont le format est décrit dans la partie « Présentation » en début d'énoncé;
- les coordonnées i et k d'une pièce dans la tour (respectivement l'indice de l'étage et la position de la pièce à cet étage);
- une liste de listes `sommes_max` ayant le même format que la liste `tour`.

Au départ, avant le premier appel de la fonction `calcul_sommes_max_rec`, tous les éléments des sous-listes de `sommes_max` sont initialisés à la valeur 0.

L'objectif de l'appel récursif `calcul_sommes_max_rec(tour, i, k, sommes_max)`, est de modifier la valeur de l'élément `sommes_max[i][k]` de la liste. Si `sommes_max[i][k]` valait 0 avant l'appel, après l'appel il contient la somme maximale des butins que l'on pourrait récupérer en suivant un chemin qui *commence* à la pièce de coordonnées (i, k) et qui va jusqu'au dernier étage de la tour.

L'appel `calcul_sommes_max_rec(tour, i, k, sommes_max)` se contente de modifier cet élément `sommes_max[i][k]`. La fonction ne renvoie rien.

II.2.a. Identifications des appels terminaux

Dans deux cas particuliers, il n'y aura pas de nouvel appel récursif de la fonction `calcul_sommes_max_rec` :

- Si l'entier `sommes_max[i][k]` est déjà différent de 0, c'est que la somme maximale en partant de la pièce de coordonnées (i, k) a été calculée dans un appel récursif précédent. Il n'y a donc plus rien à faire, la fonction peut s'arrêter sans exécuter plus d'instructions.
- Si i vaut $n-1$ (cas d'une pièce située au dernier étage de la tour), la valeur de `sommes_max[i][k]` peut être obtenue sans aucun calcul. Il n'est donc pas nécessaire de réaliser un nouvel appel récursif.

8. Compléter la **figure 3** du document-réponse en indiquant la somme maximale du butin récupérable en montant jusqu'en haut de la tour à partir des pièces situées au dernier étage de la tour (cases blanches sur le schéma de droite).

9. Compléter la définition de la fonction `calcul_sommes_max_rec` fournie dans le document-réponse en indiquant la valeur affectée à `sommes_max[i][k]` dans le cas particulier où i est égal à $n-1$.

II.2.b. Calcul récursif de `sommes_max[i][k]`

Si on connaît la somme maximale récupérable pour chacun des chemins partant des pièces de l'étage d'indice $i+1$, on peut en déduire la somme maximale pour chacun des chemins partant des pièces de l'étage d'indice i . On peut ainsi redescendre étage par étage jusqu'à obtenir la somme maximale en partant de la pièce située tout en bas de la tour.

10. À partir du résultat de la question 8, déterminer la somme maximale du butin récupérable en montant jusqu'en haut de la tour à partir des pièces situées à l'étage d'indice $i = 1$. En déduire ensuite la somme maximale récupérable en montant à partir de la pièce située à l'étage d'indice $i = 0$. Compléter la **figure 4** du document-réponse avec votre réponse.

11. Compléter la définition de la fonction `calcul_sommes_max_rec` fournie dans le document-réponse :

- en ajoutant les appels récursifs pour calculer la somme maximale en partant des deux pièces situées au-dessus de la pièce de coordonnées (i, k) ;
- en écrivant les instructions qui permettent ensuite de calculer `sommes_max[i][k]`.

II.2.c. Application

12. Définir une fonction Python d'en-tête `calcul_butin_max(tour)` qui :

- prend en argument une liste de listes `tour` contenant le butin dans chaque pièce de la tour (liste décrite dans la partie « Présentation » en début d'énoncé) ;
- crée la liste `sommes_max` (liste de listes ne contenant initialement que des 0) ;
- utilise la fonction `calcul_sommes_max_rec` pour modifier la liste `sommes_max` et calculer le butin maximal qu'il est possible de récupérer en parcourant toute la tour, du rez-de-chaussée jusqu'au dernier étage ;
- renvoie la valeur de ce butin maximal (entier).

Cette fonction nous donne le butin maximal mais pas le chemin correspondant dans la tour. En utilisant la liste `sommes_max` et un algorithme similaire à celui de la réponse 7, il serait possible de retrouver le chemin optimal (mais on ne le fera pas!).

III. Inventaire du butin

(Cette partie est totalement indépendante des parties précédentes.)

Après le succès de son expédition, Lara doit réaliser l'inventaire de tous les objets récupérés. Elle crée pour cela une liste Python dont les éléments sont les noms des objets (chaînes de caractères).

Exemple d'inventaire : `["bracelet", "statuette", "vase", "dague"]`



Tous les algorithmes écrits dans la suite devront être capable de s'adapter à des listes de tailles variables. Il n'y aura pas toujours 4 objets dans l'inventaire comme dans l'exemple précédent.

Pour pouvoir retrouver facilement les objets par la suite, on souhaite écrire un algorithme pour les trier par ordre alphabétique dans la liste.

Soit n le nombre d'objets dans la liste. L'algorithme que nous utiliserons pour la trier consiste à répéter $n - 1$ fois les opérations suivantes, pour un indice k variant de 0 à $n - 2$:

- Parmi les éléments d'indices supérieur ou égal à k de la liste, on identifie l'indice i_k du premier mot selon l'ordre alphabétique.
- On échange les valeurs des éléments d'indice k et i_k .

Exemple : Application de l'algorithme de tri à une liste de quatre éléments

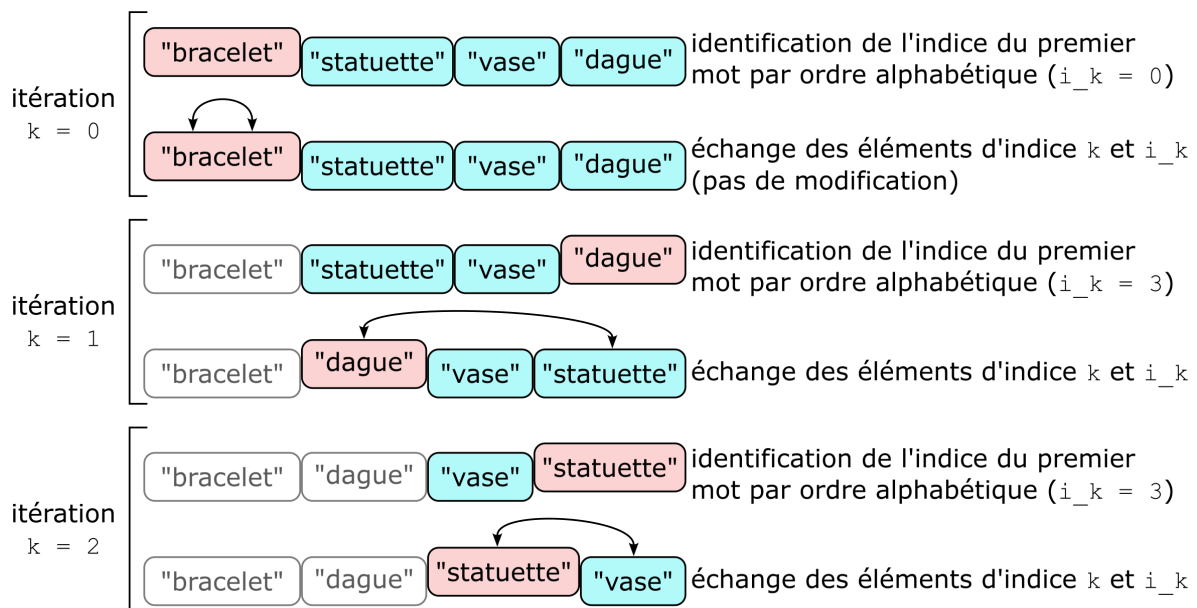


Figure n°5

Rappel : Les opérateurs < et > permettent de comparer des chaînes de caractères.

Exemple :

```
"bracelet" < "statuette"          "sceau" > "statuette"
>>> True                        >>> False
```

13. Définir une fonction Python d'en-tête `tri(inv)` qui :

- prend en argument une liste de chaînes de caractères `inv`;
- modifie la liste envoyée en arguments (sans en faire de copies) afin de trier ses éléments par ordre alphabétique.
- ne renvoie rien.

Si cette fonction utilise un autre algorithme de tri que celui décrit dans cette partie, la note 0 sera attribuée à la réponse.

14. Définir une fonction Python d'en-tête `recherche(objet, inv_trie)` qui :

- prend en argument une chaîne de caractères `objet` et une liste de chaînes de caractères `inv_trie` (les éléments de la liste `inv_trie` sont triés par ordre alphabétique);
- utilise une méthode **dichotomique** pour rechercher si la liste `inv_trie` contient un élément qui a la même valeur que celle contenue dans la variable `objet`. Si c'est le cas, la fonction doit renvoyer le booléen `True`, sinon elle renvoie le booléen `False`.

On rappelle que dans une recherche dichotomique on recherche l'élément `objet` entre les éléments d'indices `g` et `d` de la liste `inv_trie`. À chaque itération, on compare la valeur de l'élément situé au milieu de cet intervalle à la valeur de `objet`. On modifie ensuite soit `g`, soit `d` de manière à diminuer de moitié le nombre d'éléments dans l'intervalle de recherche.