Techniques de programmation

Attention : lors du chronométrage d'un algorithme, prendre bien garde à ne chronométrer que le traitement qu'il effectue et non l'affichage de son résultat. En effet, l'affichage peut être long voire conduire à un dépassement de ressources si le résultat contient des données volumineuses.

I. Diviser pour régner

1. Implémenter un algorithme de recherche linéaire **recherche_lineaire(tableau, valeur)** recherchant la position d'une valeur dans un tableau et l'appliquer à une liste triée de **10** éléments. L'algorithme doit retourner l'indice de la valeur recherchée et doit traiter le cas d'une valeur recherchée non existante.

2. Quelle est sa complexité en temps dans le pire des cas. Expliquer.

3. Implémenter un algorithme récursif de recherche dichotomique **recherche_dichotomique(tableau, valeur)**. On considérera dans un premier temps une valeur recherchée existante.

4. Améliorer l'algorithme pour traiter le cas d'une valeur recherchée non existante.

5. La complexité en temps dans le pire des cas est-elle meilleure que dans le cas de la recherche linéaire ? Expliquer.

6. Chronométrer les deux algorithmes sur une recherche dans un grand tableau, dans le pire des cas, et conclure.

7. La recherche dichotomique est-elle forcément récursive ? Justifier en développant votre réponse.

Le tri fusion est un grand classique illustrant la technique « diviser pour régner ». Pour le réaliser il suffit de savoir fusionner deux listes déjà triées.

8. Implémenter la fonction **fusion_listes(liste_1, liste_2)** fusionnant en une seule liste triée, deux listes triées **liste_1** et **liste_2**. La fonction retournera la liste fusionnée.

9. Expliquer comment fonctionne le tri fusion. Expliciter notamment le cas de base, le cas récursif en justifiant et en quoi ce tri illustre la technique « diviser pour régner ».

10. Implémenter la fonction **tri_fusion(tableau)** prenant en argument un tableau non trié et retournant le tableau trié.

11. Implémenter une version récursive de l'opération de fusion.

12. Cette version a t-elle un intérêt ? Expliquer.

II. Rotation d’un quart de tour d’une image carrée avec la technique « diviser pour régner »

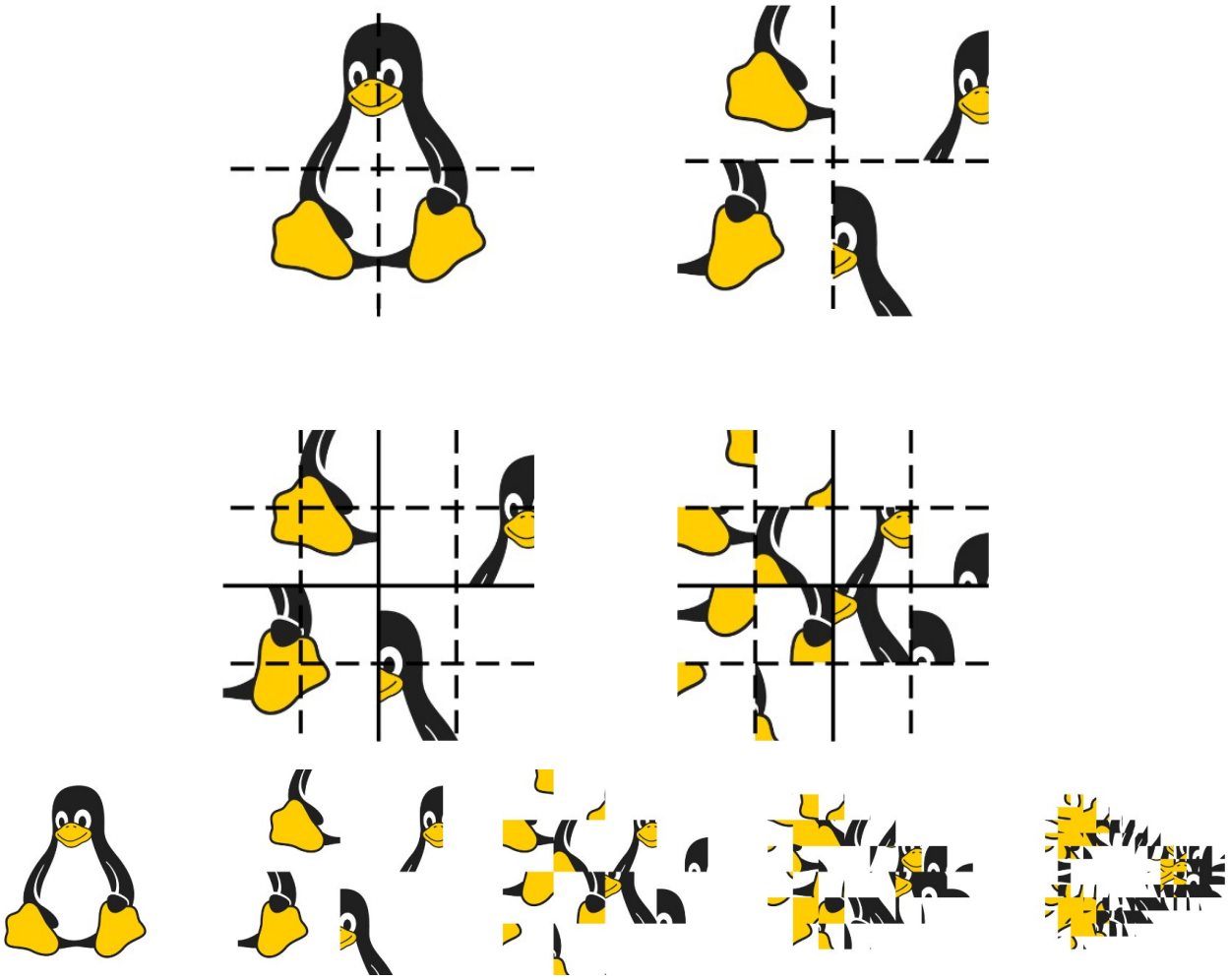
L’accès à une image et son traitement se fera avec **Open C**omputer **V**ision. **OpenCV** est une bibliothèque graphique libre, initialement développée par Intel, spécialisée dans le traitement d'images en temps réel. La société de robotique Willow Garage et la société ItSeez se sont succédé au support de cette bibliothèque. Depuis 2016 et le rachat de ItSeez par Intel, le support est de nouveau assuré par Intel (Wikipedia).

La bibliothèque écrite en C mais est utilisable sous Python grâce au module **opencv-python**. Il s’importe avec l’instruction **import cv2**. Comme les calculs en traitement d’image sont très gourmands en ressource, **opencv-python** s’utilise conjointement avec **NumPy**, nom de module **numpy**, qui s’importe avec l’instruction **import numpy as np**. C’est une bibliothèque logicielle libre et open source destinée à manipuler des matrices ou tableaux multidimensionnels ainsi que des fonctions mathématiques opérant sur ces tableaux.

1. Dans un programme **Python**, importer les modules **cv2** et **numpy** puis charger l’image **manchot.png** et l’afficher avec les fonctions **cv2.imread**, **cv2.imshow** et **cv2.waitKey**. Le programme se terminera avec la l’appel à la fonction **cv2.destroyAllWindows**.

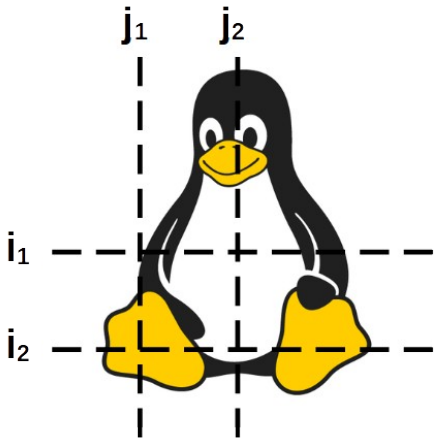
Plusieurs méthodes existent pour effectuer la rotation d’une image. La méthode proposée n’est pas la plus efficace mais est une méthode récursive intéressante à étudier.

L’idée est de diviser l’image en quatre sous-images. Il est alors aisé de les mettre à leur nouvelle place. On recommence le procédé dans chaque sous-image.



2. Quel est la cas de base ? Expliquer.

3. Décrire le cas récursif en expliquant ce qui doit être fait pour chacune des quatre parties de la sous-image considérée, 1, 2, 3 ou 4.



4. Sur quelles sous-images doit être ré-appliquée la fonction ?

Sous Python, l'image chargée avec **cv2.read** est un tableau **NumPy**. Pour extraire un sous-tableau, correspondant à une sous-image il suffit d'utiliser les crochets en précisant les lignes et les colonnes encadrant la sous-image. Par exemple **image[i1:i2, j1;j2]** extrait la sous-image entre les lignes $\mathbf{i_1}$ et $\mathbf{i_2}$ et les colonnes $\mathbf{j_1}$ et $\mathbf{j_2}$.

5. ***Attention :*** si vous avez besoin de copier temporairement une partie de tableau, il faut le faire avec la fonction **np.copy**. Sinon vous ne créerez pas un double en mémoire mais un alias. Expliquer.

6. Ecrire la fonction récursive **rotation(image)**. Et la tester.

7. Tester différents cas de base arrêtant la récursion des sous-images de taille 512 // 2, 512 // 4, 512 // 8, …

III. Programmation dynamique

1. Implémenter l'algorithme récursif **fibonacci(n)** de la suite de Fibonacci.

2. Est-ce une technique top-down ou bottom-up ? Expliquer.

3. Combien de temps met l'algorithme pour calculer les termes **n = 5, 10, 15, 20, 25, 30, 35** ? Discuter les résultats obtenus.

Pour améliorer le temps de calcul des termes de la suite, les résultats seront mémorisés dans un dictionnaire.

4. Décrire la structure de ce dictionnaire.

5. Dans une fonction **fibonacci_top_down(n)** implémenter le dictionnaire **dico_solutions_rencontrees** pour stocker les solutions de la suite de Fibonacci au fur et à mesure qu'elles sont calculées. Il n'est pas nécessaire de passer le dictionnaire en paramètre de la fonction. Pourquoi ?

6. Modifier l'algorithme afin qu'il utilise le dictionnaire pour éviter de recalculer les solutions déjà mémorisées.

7. Montrer que la mémoïsation est une technique efficace pour réduire le temps d'exécution.

8. Qu'en est-il de l'espace mémoire utilisé par l'algorithme ? Expliquer.

9. Implémenter la version bottom-up **fibonacci_bottom_up(n)** de la suite de Fibonacci.

10. Montrer que cette version permet aussi de gagner en temps d'exécution au détriment de l'espace mémoire.

11. Comparer le temps d'exécution des deux méthodes.

IV. Résolutions du problème du rendu de monnaie

1. Décrire le problème du rendu de monnaie. Comment sont choisis les systèmes monétaires ?

Les algorithmes gloutons sont très utiles lorsque trouver une seule solution à un problème donné, même non optimale, est suffisant.

En informatique, un algorithme glouton (greedy algorithm en anglais, parfois appelé aussi algorithme gourmand) est un algorithme qui recherche une solution à un problème étape par étape, par choix successifs. A chaque étape une partie du problème trouve une partie de solution.

2. Décrire l'obtention d'une solution à ce problème par algorithme glouton.

3. Ecrire une fonction **indice_plus_grande_piece_a_rendre(valeur, systeme_monetaire)** qui retourne la position de la pièce de plus grande valeur à rendre dans un système monétaire donné.

4. Implémenter l'algorithme glouton non récursif du rendu de monnaie dans une fonction **rendu_glouton(valeur, systeme_monetaire)**. Elle prend en argument un entier correspondant à la valeur à rendre et une liste d'entiers correspondant au système monétaire utilisé. Le tester sur différentes valeurs.

5. Implémenter une version récursive **rendu_glouton_rec(valeur, systeme_monetaire)** de cet algorithme glouton.

6. Sur la valeur à rendre de **14** euros, que dire des solutions obtenues pour les systèmes monétaires **[1, 2, 5, 10]**, **[1, 7, 9]** et **[2, 7, 9]** ?

En fonction du système monétaire, l'algorithme glouton précédent ne donne pas forcément la solution optimale ou même de solution. Nous allons employer la force brute pour obtenir toutes les solutions du problème.

7. Décrire une approche top-down du problème, notamment sous forme d'arbre où les branches représentent les pièces utilisées, et les nœuds les valeurs restantes après utilisation des pièces. A chaque étape, les pièces à utiliser seront parcourues de la plus petite à la plus grande. On utiliser la système monétaire **[1, 2, 5, 10]** et une somme à rendre de **4** euros.

8. Coder cette approche top-down donnant toutes les solutions du problème. La fonction sera nommée **rendu_recursif_top_down(valeur, systeme_monetaire)**. Elle retournera les listes imbriquées de pièces possibles.

9. Vérifier la solution pour la somme à rendre de 4 euros.

10. Chronométrer l'algorithme pour des sommes à rendre de 5, 10, 15, 20, 25 et 30 euros. Quelle conclusion en tirer ? Est-il possible d'améliorer l'algorithme ? Justifier.

11. Dupliquer la fonction précédente et la renommer **rendu_recursif_top_down_mem**. Implanter la mémoïsation à l'aide d'un dictionnaire nommé **solutions_deja_rencontrees**. Ses clés seront les valeurs auxquelles les solutions répondent.

12. Les performances ont-elles été améliorées ? Justifier.

13. Afin d'améliorer la lisibilité des solutions et la performance, au lieu de retourner les solutions précédemment rencontrées, retourner une chaîne de caractères incluse dans une liste de la forme **['sol X']** avec **X** la valeur à laquelle la solution répond.

14. Implémenter une version bottom-up **rendu_bottom_up(valeur, systeme_monetaire)** donnant toutes les solutions du problème sous la forme (exemple pour **valeur = 4**) :

[[[]], [[1]], [[1, 1], [2]], [[1, 1, 1], [2, 1], [1, 2]], [[1, 1, 1, 1], [2, 1, 1], [1, 2, 1], [1, 1, 2], [2, 2]]]

15. Qu'en est-il des performances de l'algorithme ?

16. Pour **valeur = 4**, mettre en évidence les chevauchements dans les solutions obtenues.

17. Comment faire pour supprimer ces chevauchements ? Coder votre solution.

V. Optimisation par sélection systématique de la meilleure solution

1. Reprendre la fonction **rendu_bottom_up** de la question II.14 et afficher les solutions pour n = 20, 25 puis 30. Que se passe-t-il ? Pouvez-vous donner facilement la meilleure solution ?

Si on ne recherche pas toutes les solutions, on peut optimiser l'algorithme afin qu'il ne recherche qu'une solution au problème, idéalement la meilleure solution. Ceci améliore ses performances et évite une analyse fastidieuse de la solution.

Dans notre exemple, à chaque étape, l'algorithme calcule toutes les solutions conduisant à la valeur à rendre correspondante. Toutes ces solutions vont être réutilisées à l'étape d'après pour calculer les solutions pour la valeur suivante.

2. Comment faire pour réduire drastiquement le nombre de solutions calculées ? Comment choisir les solutions de manière à ce que l'algorithme conduise à la meilleure solution. Expliquer.

3. Implémenter cette optimisation.

4. Vérifier que la solution obtenue est bien la meilleure pour chaque valeur calculée et que l'algorithme a amélioré la vitesse de traitement.

5. Que faire pour que l'algorithme retourne uniquement la solution pour la valeur passée en paramètre ? Le tester puis retirer la modification.

6. Tester l'algorithme avec la valeur **14** appliquée au système **[1, 7, 9]**.

7. Tester l'algorithme sur le système **[2, 7, 9]**. Que se passe-t-il ?

8. Proposer une solution à cette dernière difficulté rencontrée.