intech
ECOLE SUPERIEURE
D'INGENIERIE INFORMATIQUE

L'algorithmique des entretiens

invenietis

1 Quelques consignes

Les algorithmes doivent être implémentés avec le langage de votre choix. Chaque algorithme doit être minutieusement testé à l'aide de tests unitaires. Nous corrigerons les exercices petit à petit. Mais vous n'êtes pas obligés d'attendre une correction avant de passer à l'exercice suivant. Avancez aussi vite que possible mais, ne sautez pas d'exercice. Si un exercice vous bloque, vous devez vous acharner (et demander de l'aide). Si les consignes ne sont pas claires, faites-le savoir.

2 Strings, String Builders, Arrays, Array lists et Dictionaries

2.1 Exercice 1

Implémenter un algorithme permettant de déterminer si une chaine de caractères ne contient que des caractères uniques. Une version utilisera la ou les structures qui faciliteront le travail. Une autre n'utilisera pas de structure de données additionnelle.

2.2 Exercice 2

Ecrire un algorithme qui détermine si une chaine de caractères est une permutation d'une autre.

2.3 Exercice 3

Ecrire un algorithme qui « URLifie » les espaces d'une chaine de caractères. Mais :

- Nous ne manipulerons pas une chaine de caractères mais un tableau de caractères
- L'algorithme doit fonctionner en place
- On part du principe que le tableau initial est suffisamment grand pour supporter l'opération

2.4 Exercice 4

Implémenter un algorithme qui détermine si une chaine de caractères est une permutation d'un palindrome.

2.5 Exercice 5

On considère qu'une édition d'une chaine de caractères peut signifier l'insertion d'un caractère, la suppression d'un caractère, ou la substitution d'un caractère. Partant de cette définition, implémenter un algorithme prenant deux chaines de caractères et détermine si elles ne diffèrent que d'une édition maximum.

2.6 Exercice 6

Implémenter un algorithme de « compression » de chaines de caractères qui agrège les suites de caractères répétés. Par exemple, la chaine de caractères `agghhhssspp` sera transformé en `a1g2h3s4p2`. Si la chaine encodée est plus longue que la chaine d'origine, l'algorithme doit retourner la chaine d'origine. On partira du principe que la chaine de caractères ne contient que des lettres majuscules et minuscules.

2.7 Exercice 7

Implémenter un algorithme qui effectue une rotation de 90° sur une image carrée. L'image est représentée par une matrice de NxN pixels, chaque pixel étant encodé sur 4 bytes.

Sauriez-vous faire cet exercice en implémentant un algorithme fonctionnant en place ? Si oui, faite-le !

2.8 Exercice 8

Implémenter un algorithme prenant une matrice de NxM éléments et transforme chaque ligne et chaque colonne contenant au moins une fois l'élément 0 en une ligne ou colonne remplie de 0.

2.9 Exercice 9

Implémenter un algorithme prenant deux chaines de caractères et déterminant si l'une des deux est une rotation de l'autre. Par exemple, `lohel` est une rotation de `hello`. Vous devrez implémenter cet algorithme en faisant usage de la fonction `substring`. Mais la fonction `substring` ne devra être appelée qu'une seule fois.

3 Linked lists

3.1 Exercice 1

Ecrire un algorithme qui supprime les doublons d'une liste simplement chainée. Implémenter une première version utilisant un buffer, puis une seconde sans buffer.

3.2 Exercice 2

Implémenter un algorithme retrouvant le nième élément à partir de la fin d'une liste simplement chainée.

3.3 Exercice 3

Implémenter un algorithme qui supprime un maillon d'une liste simplement chainée en ne prenant en paramètre que ce maillon et non la tête de la liste chainée.

3.4 Exercice 4

Implémenter un algorithme partitionnant une liste chainée autour d'une valeur précisée en paramètre. C'est-à-dire que l'on veut placer au début de la liste chainée tous les éléments inférieurs à ce paramètre et à la fin de la liste chainée tous les éléments supérieurs ou égaux à ce paramètre. L'ordre dans chaque partition n'a pas d'importance.

Exemple, on veut partitionner la liste suivante en prenant la valeur 5 comme paramètre :
`3 -> 5 -> 8 -> 5 -> 10 -> 2 -> 1`
Ce qui peut nous donner par exemple :
`3 -> 2 -> 1 -> 5 -> 8 -> 5 -> 10`

3.5 Exercice 5

Dans cet exercice on représente les nombres par des listes chainées de chiffres. L'objectif est d'implémenter un algorithme capable d'additionner des nombres représentés ainsi.

Exemple, si on additionne la liste `7 -> 1 -> 6` et la liste `5 -> 9 -> 2`, l'algorithme doit retourner la liste `2 -> 1 -> 9` (617 + 295 = 912).

Dans un second temps, on implémentera un algorithme similaire mais qui travaille sur des nombres représentés « dans l'autre sens ». Exemple : En additionnant `7 -> 1 -> 6` avec `5 -> 9 -> 2`, on doit trouver la liste la liste `1 -> 3 -> 0 -> 8` (716 + 592 = 1308).

3.6 Exercice 6

Implémenter un algorithme qui indique si une liste chainée est un palindrome.

3.7 Exercice 7

Implémenter un algorithme qui détermine si deux listes simplement chainées s'intersectent et retourne le nœud d'intersection.

3.8 Exercice 8

Dans cet exercice, nous allons travailler avec une liste chainée circulaire. Il s'agit d'un cas de liste chainée corrompue dans laquelle un maillon de la liste référence un maillon qui le précède.

Par exemple la liste suivante est circulaire : `A -> B -> C -> D -> E -> C` (le même maillon C que précédemment).

Implémenter un algorithme qui retourne le maillon situé au début de la boucle. Dans le précédent exemple, il s'agit du maillon C.

4 Stacks et Queues

4.1 Exercice 1

Implémenter une stack qui en plus des opérations `push` et pop offre une opération `min` qui retourne le plus petit élément de la stack. Les 3 opérations doivent avoir une complexité en O(1).

4.2 Exercice 2

Lorsqu'une pile d'assiettes devient trop haute, elle risque de s'effondrer. Si on veut continuer d'empiler des assiettes, il faut alors commencer une nouvelle pile. Dans cet exercice, vous devez implémenter une Stack « composite » qui se comporte de la même façon. La capacité maximale de chaque « sous-stack » devra être paramétrée lors de la création de la stack composite. Les opérations push et pop de la stack composite doivent gérer de façon transparente l'utilisation des sous-stacks.

4.3 Exercice 3

Implémenter une queue dont le fonctionnement interne est basé sur 2 stacks.

4.4 Exercice 4

Implémenter un algorithme qui trie une stack de telle façon que les plus petits éléments se retrouvent au sommet de la stack. Pour cela vous pouvez utiliser une seconde stack. Aucune autre structure de données ne doit être utilisée. Seules les opérations suivantes sont disponibles sur la stack à trier et sur la stack tampon : `push`, pop, `peek` et `isEmpty`.

4.5 Exercice 5

Un refuge pour chiens et chats égarés propose aux bonnes âmes souhaitant adopter l'un d'eux les possibilités suivantes :

- Adopter le premier animal arrivé
- Adopter le premier chien arrivé
- Adopter le premier chat arrivé

Implémenter la structure de données se comportant ainsi. Elle devra exposer les opérations suivantes :

- `enqueue(Animal) : void`
- `dequeueAny() : Animal`
- `dequeueDog() : Dog`
- `dequeueCat() : Cat`

5 Trees et graphs

5.1 Exercice 1

Implémenter un algorithme qui détermine s'il existe un chemin entre deux nœuds d'un graphe orienté.

5.2 Exercice 2

Implémenter un algorithme prenant un tableau d'entiers triés uniques et trié dans l'ordre croissant et qui l'utilise pour peupler l'arbre binaire de recherche ayant la hauteur minimale.

5.3 Exercice 3

Implémenter un algorithme qui prend un arbre binaire en entrée et peuple, pour chaque niveau de l'arbre, une liste chainée à partir des éléments de ce niveau. On doit donc avoir à la sortie autant de listes chainées que de niveaux dans l'arbre.

5.4 Exercice 4

Implémenter un algorithme qui détermine si un arbre binaire est équilibré. On considère dans cet exercice qu'un arbre binaire est équilibré si la hauteur des deux sous-arbres diffère au maximum de 1.

5.5 Exercice 5

Implémenter un algorithme qui détermine si un arbre binaire est un arbre binaire de recherche.

5.6 Exercice 6

Implémenter un algorithme qui permet de trouver le successeur infixe d'un nœud dans un arbre binaire de recherche. Dans cet arbre, chaque nœud a une référence vers son parent.

5.7 Exercice 7

On dispose d'une liste de projets et d'une liste de dépendances entre ces projets (une dépendance n'étant rien d'autre qu'une paire de projets, le second dépendant du premier). Un projet ne peut être compilé que lorsque toutes ses dépendances ont été compilées.

Implémenter un algorithme qui détermine l'ordre dans lequel les projets doivent être compilés. Si aucun ordre n'est valide, l'algorithme doit retourner une erreur.

5.8 Exercice 8

Implémenter un algorithme qui détermine le premier ancêtre commun de deux nœuds d'un arbre binaire (attention, il ne s'agit pas d'un arbre binaire de recherche). Eviter de stocker des nœuds dans une ou plusieurs structures de données supplémentaires.

5.9 Exercice 9

On dispose d'un arbre binaire de recherche qui a été peuplé à partir des éléments d'un tableau parcouru de gauche à droite. Implémenter un algorithme qui construit tous les tableaux qui pourraient être la source d'un tel arbre binaire de recherche.

5.10 Exercice 10

Implémenter un algorithme qui détermine si un arbre A1 et un sous-arbre de A2.

5.11 Exercice 11

Implémenter un arbre binaire de recherche offrant les opérations suivantes :

- `insert`
- `find`
- `delete`
- `getRandomNode`

Faire en sorte que chaque nœud ait la même probabilité d'être tiré par `getRandomNode` et que cette opération soit un tant soit peu efficace.

5.12 Exercice 12

Implémenter un algorithme prenant en paramètre :

- Un arbre binaire dont chaque nœud contient un nombre entier (positif ou négatif)
- Un nombre entier (positif ou négatif) que l'on appelle N dans la suite de l'énoncé

Et qui retourne le nombre de chemins dont la somme vaut N. Un chemin n'a pas besoin de commencer à la racine ni de finir sur une feuille. Mais un parcours doit toujours avoir lieu « vers le bas » : d'un parent vers un enfant.

6 Bit manipulation

6.1 Exercice 1

Dans cet exercice nous disposons de deux nombres entiers sur 32 bits N et M et de deux positions i et j. Implémenter un algorithme qui insère M dans N afin que M commence au bit n°j et finisse au bit n°i. Un exemple sera plus clair 😊

Prenons N = 10000000000, M = 10011, i = 2, j = 6.

Alors l'algorithme doit retourner 10001001100.

6.2 Exercice 2

Implémenter un algorithme prenant un nombre réel en compris entre 0 et 1 en entrée et retourne une chaine de caractères devant correspondre à la représentation en binaire du nombre réel. S'il n'existe pas de représentation binaire précise du nombre, l'algorithme doit retourner « ERROR ».

6.3 Exercice 3

On dispose d'un entier (binaire) dans lequel on peut substituer 1 bit de 0 à 1. Implémenter un algorithme qui détermine la plus longue séquence de 1 que l'on peut créer en respectant cette simple règle.

Par exemple avec 1775 en entrée (soit 11011101111 en binaire), la plus longue séquence de 1 que l'on peut créer a une longueur de 8 (11011111111).

6.4 Exercice 4

Implémenter un algorithme prenant un entier en entrée et retourne le premier nombre supérieur et le premier nombre inférieur ayant le même nombre de 1 dans leur représentation.

6.5 Exercice 5

Implémenter un algorithme qui détermine combien de bits doivent être inversés pour passer d'un nombre A à un nombre B.

Par exemple, pour passer de 29 (11101) à 15 (01111), il faut inverser 2 bits.

6.6 Exercice 6

Implémenter un algorithme qui échange les bits paires et impaires (c'est-à-dire que les bits 0 et 1 sont échangés, ainsi que les bits 2 et 3 etc…)

6.7 Exercice 7

On représente une image monochrome à l'aide d'un tableau d'octets ce qui nous permet de stocker 8 pixels consécutifs au sein d'un octet. L'image à une largeur w divisible par 8. La hauteur de l'image peut ainsi être déduite de la taille du tableau et de w.

Implémenter une fonction qui « dessine » une ligne horizontale de (x1, y) à (x2, y).

La signature de la fonction devrait donc ressembler à :

```
drawLine(byte[] screen, int width, int x1, int x2, int y)
```

7 Récursion et programmation dynamique

7.1 Exercice 1

Imaginons un escalier constitué de n marches sur lequel on peut se déplacer en montant à chaque pas, de 1, 2 ou 3 marches. Implémenter un algorithme qui détermine le nombre de « chemins » possibles permettant de parcours l'escalier de bas en haut.

7.2 Exercice 2

Imaginons un robot situé dans le coin supérieur gauche d'une grille de r lignes et c colonnes. Le robot ne peut se déplacer que dans 2 directions : vers la droite et vers le bas. Certaines cases de la grille sont interdites.

Implémenter un algorithme qui permet au robot de se déplacer jusqu'au coin inférieur droit sans passer par les cases interdites.

7.3 Exercice 3

On appelle « index magique » dans un tableau `A[0…n-1]` un index tel que `A[i] == i`.

Implémenter un algorithme qui détermine si un tableau d'entiers distincts et triés contient un index magique et si oui, le retourne.

7.4 Exercice 4

Implémenter un algorithme qui retourne tous les sous-ensembles d'un ensemble.

7.5 Exercice 5

Implémenter un algorithme qui résout le problème classique des tours de Hanoï.

7.6 Exercice 6

Implémenter un algorithme qui retourne toutes les permutations d'une chaine de caractères (chaque caractère étant unique).

7.7 Exercice 7

Même exercice mais cette fois, les caractères ne sont pas forcément uniques.

7.8 Exercice 8

Implémenter un algorithme qui affiche toutes les combinaisons valides de n paires de de parenthèses.

Par exemple, pour n = 3, l'algorithme doit afficher : `((()))`, `(()())`, `(())()`, `()(())`, `()()()`.

7.9 Exercice 9

Implémenter la fonctionnalité de remplissage que l'on trouve sur de nombreux éditeur d'images. L'algorithme travaillera sur un tableau de couleurs.