Aix*Marseille université

Domaine Sciences et Technologies
Portail 1 René Descartes

Science informatique : TD 1

Année 2021-22

Tableaux et algorithmes numériques

Exercice 1 Un des algorithmes les plus élémentaires sur un tableau consiste à rechercher un élément dedans, à l'aide, par exemple, de la recherche séquentielle :

```
fonction rechercher_séquentiel(tableau, élément):
    n := longueur(tableau)
    Pour i de 0 à n−1 faire
        Si tableau[i] = élément alors
            retourner(i)
        FinSi
    FinPour
    retourner(−1)       # élément pas trouvé !
```

1. Modifier l'algorithme de recherche séquentielle pour qu'il renvoie vrai ou faux, selon que l'élément a été trouvé dans le tableau ou non : ainsi, la recherche de l'élément 8 dans le tableau $[3, 5, 8, 2, 8, 1]$ devra renvoyer `vrai`, alors que la recherche de 9 dans ce même tableau devra renvoyer `faux`.
2. Modifier ensuite l'algorithme pour qu'il renvoie l'indice de la *dernière occurrence* de l'élément recherché : ainsi, la recherche de l'élément 8 dans le tableau $[3, 5, 8, 2, 8, 1]$ renverra désormais 4.
3. Améliorer l'algorithme de recherche séquentielle dans le cas où le tableau donné en entrée est supposé trié, pour qu'il s'arrête dans son parcours du tableau dès lors qu'il a trouvé l'élément à chercher, ou bien qu'il est sûr que l'élément à chercher ne se trouve pas dans le tableau.
4. Quelle est l'ordre de grandeur de complexité dans le pire des cas de votre nouvel algorithme ? Comparer avec l'algorithme de recherche séquentielle non modifié.

Exercice 2 Un algorithme de tri classique est le tri par insertion, dont voici un pseudo-code possible :

```
fonction trier_par_insertion(tableau) :
    n := longueur(tableau)
    Pour i de 1 à n−1 faire
        x := tableau[i]
        # insérer x parmi les i premiers éléments
        j := i
        Tant que ((j > 0) et (x < tableau[j−1])) faire
            # décaler d'un élément
            tableau[j] := tableau[j−1]
            j := j−1
        FinTantQue
        # ici, x ≥ tableau[j−1] ou bien j= 0
        tableau[j] := x
    FinPour
    # le tableau est trié !
```

1. Rappelez la raison pour laquelle ce code ne retourne rien.
2. Que se passe-t-il si on remplace la troisième ligne de l'algorithme par `Pour i de 0 à n−1 faire` ?
3. Montrer que cet algorithme termine.
4. On a vu en cours que, dans le pire des cas, la complexité du tri par insertion est en $\mathcal{O}(n^2)$. On peut raisonnablement se poser la question de savoir si on a surestimé le nombre d'opérations élémentaires. En fait, il n'en est rien. Trouver donc une suite de tableaux $(t_n)_{n\in\mathbf{N}}$ avec t_n un tableau de longueur n telle que le nombre C_n d'opérations élémentaires effectuées lors du tri du tableau t_n est de la forme $an^2 + bn + c$ avec a, b, c des constantes et $a > 0$.

Exercice 3 L'élévation d'un nombre à une puissance entière positive est une opération de base cruciale lorsqu'on veut calculer avec des valeurs numériques. C'est même l'étape de base dans la méthode cryptographique RSA. Étant donné un nombre x (cela peut-être un entier ou un réel, qu'on représente en machine à l'aide d'un flottant) et un entier naturel n, on souhaite donc calculer le nombre $x^n = \underbrace{x \times x \times \cdots \times x}_{n \text{ fois}}$. Une première façon de calculer x^n est la fonction récursive suivante (on peut aussi écrire cette méthode à l'aide d'un algorithme non récursif, à l'aide d'une boucle) :

```
fonction exponentiation(x: entier, n: entier):
    Si n= 0 alors
        retourner 1
    Sinon
        retourner x×exponentiation(x, n−1)
    FinSi
```

1. Exécuter cet algorithme lors du calcul de 2^{10}, en précisant la valeur des variables lors de chaque appel récursif. Combien de multiplications a-t-on effectué au total pour calculer 2^{10} ?
2. Dans le cas général, combien de multiplications effectue cet algorithme pour calculer x^n, en fonction de x et n ?

   On peut réaliser l'exponentiation de manière plus rapide, en distinguant les puissances paires et impaires. Voici l'algorithme d'exponentiation rapide, en version non récursive :

```
fonction puissance(x, n):
    a := 1
    b := x
    m := n
    Tant que (m > 0) faire
        Si (m ≡ 0 mod 2) alors
            m := m/2
        Sinon
            m := (m − 1)/2
            a := a × b
        FinSi
        b := b × b
    FinTantQue
    retourner(a)
```

3. Exécuter cet algorithme lors du calcul de 2^{10}, en précisant la valeur des variables lors de chaque étape de la boucle `Tant que`.
4. Pourquoi l'algorithme termine toujours ?
5. Pour prouver que l'algorithme est correct, c'est-à-dire qu'il renvoie bien x^n, une méthode consiste à vérifier qu'à tout moment de l'algorithme on a $\mathtt{x}^{\mathtt{n}} = \mathtt{a} \times \mathtt{b}^{\mathtt{m}}$. Montrer qu'il s'agit d'un invariant de boucle (c'est-à-dire que c'est vrai avant de rentrer dans la boucle `Tant que`, puis que si c'est vrai au début d'une itération de la boucle `Tant que`, alors c'est vrai à la fin de cette itération). Conclure.
6. Les opérations élémentaires coûteuses d'un algorithme d'exponentiation (c'est le nom qu'on donne à l'*élévation à la puissance*) sont les multiplications. Combien de multiplications sont effectuées par l'algorithme d'exponentiation rapide lors du calcul de 2^{10}. Plus généralement, pouvez-vous donner un ordre de grandeur du nombre de multiplications effectuées pour calculer x^n par l'algorithme, en fonction de n ?
7. Le système de cryptographie RSA consiste à calculer des *puissances modulaires*, c'est-à-dire $x^n \mod k$, le reste de x^n dans la division euclidienne par k. Sachant que

$$\text{si } a \equiv b \mod k \qquad \text{alors } a^n \equiv b^n \mod k$$

   on a intérêt à calculer les puissances en prenant les restes dans la division euclidienne par k à chaque étape. En vous inspirant de la fonction `puissance`, en ajoutant un troisième argument k, proposer une fonction **récursive** qui calcule $x^n \mod k$: on s'autorise à utiliser

comme opération élémentaire supplémentaire le calcul de $y \mod k$, le reste dans la division euclidienne de y par k.

Exercice 4 Dans l'armée de Sparte (autour du Vème siècle avant J.-C.), les militaires étaient parfois amenés à se transmettre des messages chiffrés. Pour ce faire, ils utilisaient un bâton, appelé bâton de Plutarque (ou scytale). L'émetteur du message prenait une fine lanière de tissu ne pouvant contenir qu'une seule lettre dans sa largeur, l'enroulait en spirale autour d'un bâton, et écrivait son message ligne par ligne dans la longueur du bâton de façon à avoir toutes les lignes remplies sauf éventuellement la dernière ligne :

Une fois déroulée, la lanière contenait le message chiffré. Pour déchiffrer ce message, le récepteur devait posséder un bâton de même diamètre (ayant le même nombre de circonvolutions) que celui de l'émetteur. Ce type de codage est un chiffrement par transposition.

À titre d'exemple, considérons le message suivant : « La vie c'est comme une boîte de chocolats, on ne sait jamais sur quoi on va tomber. » (*Forrest Gump*, de Robert Zemeckis) Considérons un bâton dont le périmètre permet 6 circonvolutions (*nous dirons pour simplifier que la clé du chiffrement est* 6). Le message chiffré est alors « Lo␣␣moamdoan␣meni␣ve␣␣svi␣cn␣aeuhes␣␣no␣utcecsro'␣oa␣mebliqbsoatuetît␣or␣tsji.ce,a␣␣ »

1. Chiffrer la phrase « Les cons ça ose tout. » (*Les tontons flingueurs*, Michel Audiard) avec la clé 4 en vous aidant d'un ruban de papier que vous enroulerez autour d'un stylo par exemple. (C'est plus facile à faire à deux...)
2. Une matrice s'impose naturellement si on souhaite automatiser ce procédé. Décrire le nombre de lignes et de colonnes à utiliser pour chiffrer un message m de longueur n avec une clé c.
3. Compléter le pseudo-code suivant permettant de chiffrer un message m avec la clé c :

```
fonction chiffrement_spartiate(m: tableau de caractères, c: entier):
    n := longueur(m)
    A := matrice vide de [          ] lignes et [          ] colonnes
    Pour i de 0 à [          ] faire
        Pour j de 0 à [          ] faire
            l := [                                ]
            Si l < n alors
                A[i][j] := [          ]
            Sinon
                A[i][j] := ␣
            FinSi
        FinPour
    FinPour
    message_chiffré := tableau vide de longueur n
```

```
retourner message_chiffré
```

4. Étant donné un message chiffré m de longueur n et une clé c, donner un algorithme permettant de découvrir le message d'origine.
5. Utiliser la méthode précédente pour déchiffrer les messages suivants (on peut s'aider d'une implémentation en Python pour aller plus vite !) :
   — "Ce'e'␣oceànos␣␣ntçln␣aeam␣siêq␣tmur." avec $c = 4$;
   — "Càq␣s'␣up␣emea?so␣r␣titl␣␣␣ue␣" avec $c = 5$;
   — "L,␣snru␣s␣␣ekjutp.eeioè␣" avec $c = 6$.
6. À présent, vous recevez un message chiffré m sans la clé de chiffrement. Proposer une méthode permettant de retrouver le message d'origine et en évaluer la complexité en temps.