Science informatique : test diagnostique

Durée : 45 minutes

1. Quel est l'entier positif dont l'écriture binaire (sans bit de signe) est 10110 ?

   Solution : L'entier correspondant est $0 \times 2^0 + 1 \times 2^1 + 1 \times 2^2 + 0 \times 2^3 + 1 \times 2^4 = 2 + 4 + 16 = 22$.

2. Combien de nombre entiers différents peut-on coder sur 7 bits ?

   Solution : Chaque bit supplémentaire multiplie par deux le nombre de codes possibles. Avec un bit on a $2 = 2^1$ codes, avec deux bits on a donc $2 * 2 = 2^2$ codes, avec trois $2 * 2^2 = 2^3$ codes, etc. En continuant ce raisonnnement jusqu'à sept, on obtient qu'avec sept bits on peut coder $2^7 = 128$ nombres entiers différents.

3. Combien d'octets sont contenus dans 6,4 Mb (mégabits) ? Exprimer le résultat en kilooctets. (On utilisera la convention que 1Mb= 10^6b et 1Ko= 10^3o.)

   Solution : Puisqu'un octet vaut 8 bits, 6,4 Mb contiennent $6{,}4/8 = 0{,}8$ Mo = 800 ko (kilooctets).

4. Considérer l'algorithme suivant, écrit dans un pseudo-code proche de Python :

```
1  fonction mystère(A : tableau d'entiers):
2      n := longueur(A)
3      B := tableau de longueur n rempli de 0
4      Pour i de 0 à n−1 faire
5          j := 0
6          Tant que j < n faire
7              B[i] := B[i] + A[j]
8              j := j + 2
9          FinTantQue
10         B[i] := B[i] × A[i]
11     FinPour
12     retourner B
```

   L'exécuter sur le tableau `[2, 1, 3, 2]` en donnant toutes les valeurs des variables pendant l'exécution et le résultat.

   Solution : Le résultat est $[10, 5, 15, 10]$, et les valeurs des variables évoluent comme dans la table suivante :

A	n	B	i	j
[2, 1, 3, 2]	4	[0, 0, 0, 0]	0	0
		[2, 0, 0, 0]		2
		[5, 0, 0, 0]		4
		[10, 0, 0, 0]	1	0
		[10, 2, 0, 0]		2
		[10, 5, 0, 0]		4
		[10, 5, 0, 0]	2	0
		[10, 5, 2, 0]		2
		[10, 5, 5, 0]		4
		[10, 5, 15, 0]	3	0
		[10, 5, 15, 2]		2
		[10, 5, 15, 5]		4
		[10, 5, 15, 10]		

5. Calculer, en justifiant rapidement, le nombre d'instructions effectuées par l'algorithme `mystère` ci-dessus en fonction de la longueur n du tableau donné en argument. Vous pourrez simplifier le résultat en utilisant la notation « grand O », si vous la connaissez.

Solution : On exécute les lignes 2, 3 et 12 de l'algorithme seulement une fois, pour un total de 3 instructions.
On exécute le contenu de la boucle qui commence à la ligne 4 n fois ; la ligne 4 elle-même est exécutée $n+1$ fois (on sort de la boucle quand $i = n$, donc une fois de plus). Donc on exécute n fois les lignes 5 et 10.
Comme la variable j est augmentée de 2 à chaque tour de la boucle « tant que », on n'exécute les instructions des lignes 7 et 8 approximativement $n/2$ fois pour chaque tour de la boucle « pour », et la ligne 6 une fois de plus, donc $n/2+1$, ce qui donne $3n/2+1$ exécutions des lignes 6 à 8 pour chaque tour de la boucle « pour ».
Donc, dans la boucle « pour », on exécute n fois multiplié par $3n/2+1+2 = 3n/2+3$ fois les lignes de 5 à 10, ce qui donne $n \times (3n/2+3) = 3n^2/2+3n$, et il faut ajouter $n+1$ exécutions de la ligne 4, ce qui donne $3n^2/2+4n+1$ instructions.
En ajoutant les lignes 2, 3 et 12 on obtient $3n^2/2+4n+4$, qu'on peut simplifier en $\mathcal{O}(n^2)$.

6. Compléter le pseudo-code suivant permettant de chercher si un élément `e` est présent dans un tableau `A` trié dans l'ordre croissant, à l'aide d'une recherche dichotomique :

```
1  fonction rechercher_dichotomique(tableau, e) :
2      n := longueur(tableau)
3      début := 0
4      fin := n−1
5      Tant que [                    ] faire
6          milieu := [                              ]
7          Si tableau[milieu] = e alors
8              retourner(milieu)
9          Sinon Si tableau[milieu] < e alors
10             [                    ]
11         Sinon
12             [                    ]
13         FinSi
14     FinTantQue
15     retourner(−1)
16
```

Solution :

```
1  fonction rechercher_dichotomique(tableau, élément) :
2      n := longueur(tableau)
3      début := 0
4      fin := n−1
5      Tant que début ≤ fin faire      # on s'arrête lorsque la portion est vide
6          milieu := ⌊(début + fin)/2⌋    # calcul du milieu
7          Si tableau[milieu] = élément alors
8              retourner(milieu)
9          Sinon Si tableau[milieu] < élément alors
10             début := milieu+1        # l'élément est à droite
11         Sinon
12             fin := milieu−1          # l'élément est à gauche
13         FinSi
14     FinTantQue
15     retourner(−1)
```

7. On considère le tri par insertion et le tri par fusion. Quelles sont leurs complexités respectives, en fonction de la longueur n du tableau à trier ?

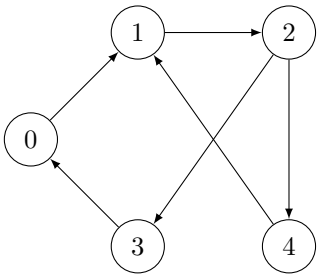
Solution : $\mathcal{O}(n^2)$ et $\mathcal{O}(n\log(n))$

8. Proposer un algorithme `décroissant(A)` (en pseudo-code ou en Python) qui prend en entrée un tableau d'entiers `A` et renvoie `vrai` (`True` en Python) si le tableau est trié *en ordre décroissant*, et `faux` (`False` en Python) autrement. Par exemple, `décroissant([2, 1, -1])` renvoie `vrai`, alors que `décroissant([2, 0, 1])` renvoie `faux`.

Solution :

```
fonction décroissant(A : tableau d'entiers):
    n := longueur(A)
    i := 0
    Pour i := 0 à n - 2 faire        # on s'arrête avant d'arriver à n - 1
        Si A[i] < A[i+1] alors
            retourner faux
        FinSi
    FinPour
    retourner vrai
```

9. Donner la matrice d'adjacence du graphe orienté représenté ci-dessous :



Solution : $\begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{pmatrix}$

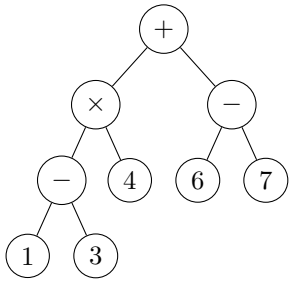
10. Citer un cycle du graphe ci-dessus.

Solution : Il existe deux cycles sans répétition, à permutation près : $0, 1, 2, 3, 0$ et $1, 2, 4, 1$.

11. Qu'obtient-on en exécutant un parcours en profondeur du graphe depuis le sommet 2 ?

Solution : Il y a deux parcours en profondeur possibles. En partant du sommet 2, on peut visiter d'abord le voisin 3, puis son voisin 0, puis son voisin 1. On remonte alors à l'unique voisin manquant de 2, le sommet 4. L'autre possibilité produit le parcours 2 4 1 3 0.

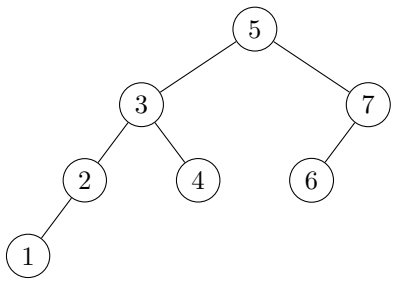
12. Donner la liste des valeurs des nœuds traversés si on effectue un parcours postfixe (ou suffixe) de l'arbre binaire suivant, représentant une expression arithmétique.



Solution : On obtient la notation polonaise inversée (non parenthésée) de l'expression arithmétique : 1 3 − 4 × 6 7 − +

13. La propriété fondamentale des arbres binaires de recherche est que, pour chaque nœud x de l'arbre :
— tous les nœuds dans le sous-arbre gauche ont une valeur inférieure à la valeur de x ;
— tous les nœuds dans le sous-arbre droit ont une valeur supérieure à la valeur de x.
Construire un arbre binaire de recherche en insérant une par une les valeurs 5, 7, 6, 3, 4, 2, 1 dans l'ordre indiqué.

Solution :



14. Proposer un algorithme récursif (en pseudo-code ou en Python) prenant en argument deux entiers positifs x et n, renvoyant la puissance x^n, en distinguant le cas d'une puissance n nulle ou strictement positive. On s'attend à ce que la fonction réalise un nombre linéaire d'appels récursifs, en fonction de la puissance n.

Solution :

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

15. Citer un algorithme utilisant la méthode diviser-pour-régner.

Solution : Parmi les algorithmes au programme de NSI, utilisant la méthode diviser-pour-régner, citons la recherche dichotomique et le tri fusion, par exemple.

16. Pourquoi l'algorithme suivant, calculant le n-ième terme de la suite de Fibonacci, n'est-il pas intéressant en pratique ? Comment l'améliorer ?

```
1  fonction fibonacci(n: entier):
2      Si (n= 0 ou n= 1) alors
3          retourner 1
4      Sinon
5          retourner fibonacci(n−1) + fibonacci(n−2)
```

Solution : L'algorithme ci-dessus est un algorithme récursif, qui réalise de nombreux appels récursifs en de multiples occurrences. En particulier, il est de complexité exponentielle et n'est donc pas utilisable en pratique. On lui préfère une méthode de programmation dynamique consistant à stocker les résultats dans un tableau (ici, on pourrait se contenter de stocker les deux derniers termes de la suite) :

```
1  fonction fibonacci_amélioré(n: entier):
2      tableau := tableau de longueur n+1 rempli de 1
3      Pour i de 2 à n faire
4          tableau[i] := tableau[i−1] + tableau[i−2]
5      FinPour
6      retourner tableau[n]
```