Aix*Marseille université

Domaine Sciences et Technologies
Portail 1 René Descartes

Introduction à l'informatique : TD 4

Code UE : S14IN1I1

Année 2021–2022

Algorithmes et tableaux

Les algorithmes peuvent nous permettre de compter sans effort. Par exemple, pour calculer la somme des entiers de 1 à n qui sont divisibles par 3 ou par 5, on a vu en cours, on peut définir une fonction `sommer_divisible_3_5` prenant en argument un entier (on peut préciser le *type* de l'argument et du retour, mais c'est tout à fait optionnel) :

```
fonction sommer_divisible_3_5(n: entier) -> entier:
    somme := 0
    Pour i de 1 à n faire
        Si (i ≡ 0 mod 3) ou (i ≡ 0 mod 5) alors
            somme := somme + n
        FinSi
    FinPour
    retourner(somme)
```

On peut ensuite appeler cette fonction à l'aide `sommer_divisible_3_5(1000)` si on veut calculer la somme des entiers de 1 à 1000 qui sont divisibles par 3 ou par 5.

Exercice 1 (Question de somme) On cherche à calculer la somme des n premières puissances de 2. Par exemple, si $n = 6$, le résultat est $2^0+2^1+2^2+2^3+2^4+2^5 = 1+2+4+8+16+32 = 63$.

1. Écrire une première fonction réalisant ce calcul, en s'autorisant le calcul des puissances de 2 comme opération élémentaire (c'est-à-dire qu'on peut écrire 2^i dans le code), écrire un algorithme calculant et affichant la somme des n premières puissances de 2.
2. Combien d'opérations élémentaires effectue votre fonction lorsqu'elle est appelée avec un entier n en argument ?
3. Comment modifier votre code si on ne s'autorise plus l'utilisation de 2^i comme opération élémentaire ? Calculer à nouveau le nombre d'opérations en fonction de n.
4. En calculant explicitement la valeur de $S = 2^0 + 2^1 + \cdots + 2^{n-1}$ en déduire une façon de calculer S avec un seul calcul de puissance et un nombre constant d'opérations élémentaires (additions, soustractions, multiplications...) supplémentaires.

En cours, les algorithmes nous ont permis de recherche un élément dans un tableau, à l'aide, par exemple, de la recherche séquentielle dans un tableau :

```
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 !
```

Exercice 2

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. Rappeler la complexité dans le pire des cas de l'algorithme de recherche séquentielle donné plus haut, en terme de l'ordre de grandeur du nombre d'opérations élémentaires en fonction de la longueur n du tableau.

4. 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.
5. Quelle est l'ordre de grandeur de complexité dans le pire des cas de votre nouvel algorithme ?

On a aussi vu en cours que les algorithmes permettaient de jouer... par exemple au dé. On se donne une fonction `alea` qui prend en argument deux entiers a et b et retourne un nombre entier choisi aléatoirement dans l'intervalle $[a; b]$. Alors, il est possible de faire afficher à l'utilisateur le nombre d'étapes avant d'avoir obtenu le premier 5 en lançant successivement le dé grâce à l'algorithme suivant :

```
nombre_étapes := 0
face_dé := alea(1, 6)
Tant que (face_dé ≠ 5) faire
   face_dé := alea(1, 6)
   nombre_étapes := nombre_étapes + 1
FinTantQue
écrire("Le nombre d'étapes avant le premier 5 était ", nombre_étapes)
```

Noter que dans l'appel à la fonction `écrire`, on s'est autorisé de mettre deux arguments plutôt qu'un : plus généralement, on peut y mettre autant d'arguments que l'on veut, la fonction imprimant à l'utilisateur les différents messages les uns à la suite des autres. On peut aussi modifier le code précédent pour remplacer la face 5 par une face choisie par l'utilisateur : on peut le lui demander en début d'algorithme à l'aide de l'appel à

```
écrire("Quelle face du dé voulez-vous tester ?")
face_dé := lire()
```

Exercice 3 (Le juste prix)

1. Écrire un algorithme qui :
   — choisit aléatoirement un nombre entier entre 0 et 1000 de manière parfaitement opaque pour l'utilisateur ;
   — demande à l'utilisateur d'entrer des nombres entiers jusqu'à ce que ce dernier trouve le nombre préalablement choisi, en indiquant à chaque tentative « trop haut » ou « trop bas » selon le nombre saisi au clavier.
2. Comment modifier votre algorithme pour qu'il affiche à l'utilisateur le nombre de tentatives qu'il a utilisé, une fois le nombre trouvé ?
3. Dans le pire des cas, en combien d'étapes pouvez-vous être sûr que vous aurez trouvé le nombre mystère ? Quel est ce nombre d'étapes lorsqu'on cherche un nombre mystère entre 0 et un entier naturel n de la forme $2^p - 1$? Et pour un entier naturel n quelconque ?

Exercice 4 (Problème de pesée.s)

1. On dispose de 6 pièces de 1 euro dont une seule est fausse et plus lourde que les autres. Montrer qu'on peut la détecter en utilisant une balance de Roberval (balance à plateaux comme celle ci-dessous) et en ne procédant qu'à 2 pesées.
2. En combien de pesées est-il possible de distinguer la fausse pièce de manière certaine parmi un ensemble de 9 pièces ?
3. Trouver ces résultats pour des nombres de pièces spécifiques, c'est bien. Mais il est beaucoup plus intéressant de connaître et de démontrer la réponse dans le cas général. Montrer que, si le nombre de pièces p appartient à l'intervalle $]3^{n-1}; 3^n]$, alors il est possible de retrouver la pièce fausse en n pesées.