Aix*Marseille université

PORTAIL ET PES 1 RENÉ DESCARTES

Introduction à l'informatique : Partiel
Code UE : S14IN1I1

Année 2020-21

Documents et calculatrice interdits

Tous les exercices sont indépendants et peuvent donc être traités dans n'importe quel ordre. Le sujet est long (et le barème de correction en tiendra compte) : vous pouvez donc passer une question vous semblant trop difficile, quitte à y revenir ensuite s'il vous reste du temps. Par ailleurs, toute réponse non correctement rédigée (avec explication précise et concise) ou non justifiée sera considérée comme fausse.

Exercice 1 (codage des nombres)

1. Donnez le codage en binaire de l'entier naturel $(338)_{10}$, en précisant brièvement (5 lignes maximum) votre méthode.

Solution : Il existe 2 méthodes :

— On commence par retrouver les puissances de 2 successives :

n	0	1	2	3	4	5	6	7	8
2^n	1	2	4	8	16	32	64	128	256

Dans $(338)_{10}$, on peut placer $(256)_{10} = 2^8$ comme plus grande puissance de 2. Il reste $(338 - 256)_{10} = (82)_{10}$. Dans $(82)_{10}$, on peut placer $(64)_{10} = 2^6$ de la même manière. Il reste $(82-64)_{10} = (18)_{10}$. Dès lors, il est aisé de voir que $(18)_{10} = (16)_{10} + (2)_{10} = 2^4 + 2^1$. Par conséquent, on peut écrire que :

$$\begin{aligned}(338)_{10} &= (256)_{10} + (64)_{10} + (16)_{10} + (2)_{10}\\ &= 2^8 + 2^6 + 2^4 + 2^1\\ &= \mathbf{1} \times 2^8 + \mathbf{0} \times 2^7 + \mathbf{1} \times 2^6 + \mathbf{0} \times 2^5\\ &\quad +\mathbf{1} \times 2^4 + \mathbf{0} \times 2^3 + \mathbf{0} \times 2^2 + \mathbf{1} \times 2^1 + \mathbf{0} \times 2^0.\end{aligned}$$

Ainsi, l'écriture en base 2 (en binaire) de l'entier naturel $(338)_{10}$ est $(101010010)_2$.

— On fait une division euclidienne de $(338)_{10}$ par 2, on note le quotient et le reste et on réitère le processus sur le quotient, jusqu'à obtenir un quotient égal à 0. Ensuite, il suffit d'écrire les restes dans l'ordre inverse de leur obtention. On a donc :

$$\begin{aligned}(338/2)_{10} &= (169 + \mathbf{0})_{10}\\ (169/2)_{10} &= (84 + \mathbf{1})_{10}\\ (84/2)_{10} &= (42 + \mathbf{0})_{10}\\ (42/2)_{10} &= (21 + \mathbf{0})_{10}\\ (21/2)_{10} &= (10 + \mathbf{1})_{10}\\ (10/2)_{10} &= (5 + \mathbf{0})_{10}\\ (5/2)_{10} &= (2 + \mathbf{1})_{10}\\ (2/2)_{10} &= (1 + \mathbf{0})_{10}\\ (1/2)_{10} &= (0 + \mathbf{1})_{10},\end{aligned}$$

soit $(101010010)_2$.

2. Donnez l'entier naturel en base 10 dont le code binaire est $(11010111)_2$, en précisant brièvement (5 lignes maximum) votre méthode.

Solution : On lit le nombre de la droite vers la gauche (des bits de poids faible vers les bits de poids fort). Le $i$ème chiffre le composant dans cet ordre correspond au coefficient de la $(i-1)$ème puissance de 2 qui entre dans la somme des puissances de 2 qui permet de calculer le nombre. Nommons $(x)_{10}$ le nombre en base 10 à trouver. On a donc :

$$\begin{aligned}(x)_{10} &= (11010111)_2\\ &= (1 \times 2^0 + 1 \times 2^1 + 1 \times 2^2 + 0 \times 2^3 + 1 \times 2^4 + 0 \times 2^5 + 1 \times 2^6 + 1 \times 2^7)_{10}\\ &= (1 \times 2^0 + 1 \times 2^1 + 1 \times 2^2 + 1 \times 2^4 + 1 \times 2^6 + 1 \times 2^7)_{10}\\ &= (1 + 2 + 4 + 16 + 64 + 128)_{10}\\ &= (215)_{10}.\end{aligned}$$

3. Donnez en base 10 le nombre réel en virgule flottante codé en binaire par les 32 bits suivants : 01011001 11101100 00000000 00000000.

Solution : Pour rappel, un nombre réel en virgule flottante est représenté en base 2 par 32 bits tels que :

- le 1er bit représente le signe s : 0 représente + et 1 représente −.
- les 8 bits suivants encodent l'exposant k. k est obtenu en soustrayant $(127)_{10}$ au nombre entier en base 10 encodé en base 2 par ces 8 bits.
- les 23 bits suivants encodent la mantisse m telle que $m \in [1; 2[$ (le chiffre avant la virgule n'est pas encodé et vaut toujours 1, ce qui induit que les 23 bits servent à encoder les chiffres après la virgule).

Pour notre exercice, à présent :

- le 1er bit est 0, ce qui indique que le nombre recherché est positif. Autrement dit, $s = +$;
- les 8 bits qui suivent sont : 10110011. On obtient l'exposant k ainsi :

$$\begin{aligned} k &= (10110011)_2 - (127)_{10} \\ &= (\mathbf{1} \times 2^7 + \mathbf{1} \times 2^5 + \mathbf{1} \times 2^4 + \mathbf{1} \times 2^1 + \mathbf{1} \times 2^0 - 127)_{10} \\ &= (128 + 32 + 16 + 2 + 1 - 127)_{10} \\ &= (179 - 127)_{10} \\ &= (52)_{10}. \end{aligned}$$

- les 23 bits qui suivent sont : 110110...0. On obtient la mantisse m ainsi :

$$\begin{aligned} m &= (1 + \mathbf{1} \times \tfrac{1}{2^1} + \mathbf{1} \times \tfrac{1}{2^2} + \mathbf{0} \times \tfrac{1}{2^3} + \mathbf{1} \times \tfrac{1}{2^4} + \mathbf{1} \times \tfrac{1}{2^5})_{10} \\ &= (\tfrac{2^5+2^4+2^3+2^1+2^0}{2^5})_{10} \\ &= (\tfrac{32+16+8+2+1}{32})_{10} \\ &= (\tfrac{59}{32})_{10} \\ &= (1,84375)_{10}. \end{aligned}$$

Le nombre recherché est donc $(1,84375 \times 2^{52})_{10}$, ou $(\frac{59}{32} \times 2^{52})_{10}$.

Exercice 2 (nombres parfaits)

Soit n un nombre entier naturel. Le nombre $d \in \mathbb{N}$ est un diviseur de n si $n \bmod d = 0$, autrement dit si le reste de la division de n par d est nul. Un nombre entier naturel est *parfait* s'il est égal à la somme de ses diviseurs stricts (différents de lui-même). L'algorithme suivant permet de déterminer si un nombre entier naturel est parfait :

```
Fonction nb_parfait(n: entier): booléen
Début
   Si (n = 0) alors
      retour faux;
   Sinon
      s := 1;
      d := 2;
      Tant que d ≤ ⌊n/2⌋ faire     # ⌊n/2⌋: quotient de la division euclidienne de n par 2
         Si (n mod d = 0) alors
            s := s + d;
         Fin si
         d := d + 1;
      Fin faire
      Si (n ≠ s) alors
          retour faux;
      Sinon
          retour vrai;
      Fin si
   Fin si
Fin
```

1. Exécutez l'algorithme `nb_parfait` sur l'entier $n = 8$, en montrant toutes les valeurs des variables pendant l'exécution, le résultat et en comptant le nombre de tours de boucle (ou étapes d'itération) effectués. L'entier $n = 8$ est-il parfait ?

Solution : Pour l'entrée $n = 8$ on a les valeurs suivantes des variables pendant l'exécution :

n	tour de boucle	d	s
8	n/a	n/a	1
	1	2	3
	2	3	3
	3	4	7

Au sortir de la boucle, au bout de 3 tours, $n \neq s$, donc 8 n'est pas un nombre parfait.

Dans le tableau ci-dessus, les lignes indiquant des tours de boucle valides donnent les valeurs des variables d et s à la fin de l'étape d'itération.

2. Même question pour l'entier $n = 28$.

Solution : Pour l'entrée $n = 28$ on a :

n	tour de boucle	d	s
28	n/a	n/a	1
	1	2	3
	2	3	3
	3	4	7
	4	5	7
	5	6	7
	6	7	14
	7	8	14
	8	9	14
	9	10	14
	10	11	14
	11	12	14
	12	13	14
	13	14	28

Au sortir de la boucle, au bout de 13 tours, $n = s$, donc 28 est un nombre parfait.

3. Expliquez pourquoi l'algorithme `nb_parfait` termine pour toute entrée $n \geq 0$.

Solution : Dans le cas où $n = 0$, l'algorithme s'arrête tout de suite en renvoyant faux. Dans le cas où $n > 0$, la variable d a initialement la valeur 2 puis est incrémentée de la valeur 1 à chaque tour de boucle jusqu'à atteindre la valeur $\lfloor n/2 \rfloor + 1$, qui marque la terminaison de la boucle. De par l'incrémentation de la valeur 1 à chaque tour de boucle, d atteindra nécessairement cette valeur, ce qui induit que l'algorithme termine.

4. Calculez, en justifiant, le nombre d'opérations élémentaires effectuées par l'algorithme `nb_parfait` en fonction de la valeur de n, prise quelconque. Vous pourrez simplifier le résultat en utilisant la notation « grand $\mathcal{O}$ ».

Solution : Dans le cas où $n = 0$, une seule opération est réalisée (le `retour faux`) et l'algorithme termine. Dans le cas où $n > 0$, les opérations réalisées sont les suivantes :

- En dehors de la boucle, on trouve :
  - 2 affectations avant la boucle $\quad \to$ 2 opérations ;
  - 1 test conditionnel et 1 retour après la boucle $\quad \to$ 2 opérations.
- Au sein de la boucle qui est exécutée $\lfloor n/2 \rfloor - 1$ fois, on trouve, dans le pire cas :
  - 1 test conditionnel (celui du tant que) fondé sur 1 division entière $\quad \to$ 2 opérations ;
  - 1 test conditionnel fondé sur une opération de modulo $\quad \to$ 2 opérations ;
  - dans le cas du test valide (admettons qu'il est valide chaque fois dans le pire cas), 1 affectation et une addition $\quad \to$ 2 opérations ;
  - l'incrémentation de d, ce qui nécessite 1 affectation et une addition $\quad \to$ 2 opérations ;

Par conséquent, dans le pire cas, on a 4 opérations réalisées en dehors de la boule et 8 opérations réalisées à l'intérieur de la boucle qui est itérée $\lfloor n/2 \rfloor - 1$ fois. Par conséquent, le nombre d'opérations élémentaires exécutées est de :

$$\begin{aligned} & 8 \times (\lfloor n/2 \rfloor - 1) + 4 \\ = \; & \lfloor 8n/2 \rfloor - 8 + 4 \\ = \; & 4n - 4. \end{aligned}$$

Le nombre d'opérations élémentaires est donc linéaire, ce que l'on peut simplifier en disant que l'algorithme `nb_parfait` est en $\mathcal{O}(n)$.

Exercice 3 (palindromes)

Un palindrome est un mot dont la première lettre est identique à la dernière, la deuxième à l'avant-dernière, etc. À titre d'exemple, le mot « non », tout comme « esse » ou encore « ressasser » sont des palindromes de la langue française. Au delà de la langue française, le concept de palindrome s'étend à toutes les langues, mais aussi à toute suite de symboles d'un alphabet donné. Ainsi, le nombre entier 153 en base 10 devient un palindrome en base 2 : 10011001. La question « Was it a car or a cat I saw ? », est également un palindrome si l'on ne considère ni les espaces ni le point d'interrogation. Dans cet exercice, nous voulons écrire un algorithme qui décide si une chaîne de caractères représentée par un tableau de caractères est un palindrome, c'est-à-dire un algorithme qui répond vrai ou faux pour un mot donné en paramètre.

1. Écrivez en pseudo-code cet algorithme, dont le prototype est le suivant :

   `Fonction palindrome(C[0..`$n-1$`]: tab_caractères): booléen`,

   où C[0..$n-1$] représente un tableau de caractères de taille n. Prenez soin d'expliquer le principe de votre algorithme, en argumentant sa pertinence dans un texte court (une dizaine de lignes) le précédant.

   Solution : D'après la définition d'un palindrome, pour déterminer si la chaîne de caractère C donnée en est un, il va falloir comparer des lettres le composant deux à deux. Par hypothèse, n est la longueur de la chaîne, nous allons comparer successivement : C[0] à C[$n-1$], C[1] à C[$n-2$]... et plus généralement C[i] à C[$n-i-1$]. Nous allons utiliser une boucle avec un indice i qui effectuera ces comparaisons. Si l'on faisait varier i de 0 à $n-1$, les comparaisons seraient faites deux fois (à l'exception du caractère au milieu dans le cas d'une chaîne de longueur impaire). On en déduit qu'il suffit que l'indice i ne parcourt que la moitié du mot, à savoir $0 \leq i \leq \lfloor n/2 \rfloor - 1$ (le -1 sert à éviter de parcourir le caractère du milieu dans le cas d'une chaîne de longueur impaire). Enfin, on va faire en sorte de ne procéder qu'à des tests nécessaires et suffisants en s'arrêtant dès le constat d'une comparaison invalide. L'algorithme peut ainsi s'écrire :

   ```
   Fonction palindrome(C[0..n − 1]): booléen
   Début
      i := 0;
      Tant que (i ≤ ⌊n/2⌋ − 1) faire
         Si (C[i] ≠ C[n − i − 1]) alors
            retour faux;
         Fin si
         i := i + 1;
      Fin faire
      retour vrai;
   Fin
   ```

2. Exécutez l'algorithme `palindrome` sur le mot « ressasser » en montrant toutes les comparaisons effectuées.

   Solution : Pour l'entrée « ressasser », on a :

C	tour de boucle	i	C[i]	C[$n-i-1$]
ressasser	1	0	'r'	'r'
	2	1	'e'	'e'
	3	2	's'	's'
	4	3	's'	's'

   Au sortir de la boucle, au bout de 4 tours, toutes les comparaisons sont valides et l'algorithme décide que « ressasser » est bien un palindrome.

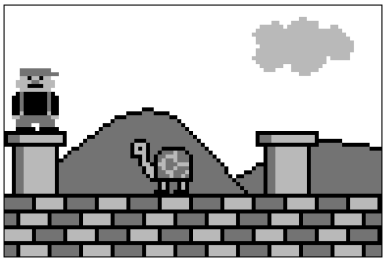
3. Évaluez la complexité en temps de votre algorithme. Vous calculerez pour ce faire précisément le nombre d'opérations élémentaires que vous simplifierez en utilisant la notation « grand $\mathcal{O}$ ».

   Solution : Les opérations réalisées sont les suivantes :
   - En dehors de la boucle, on a 1 affectation et 1 retour → 2 opérations.
   - Tout d'abord, relevons que le pire cas se produit quand le mot est effectivement un palindrome. À l'intérieur de la boucle qui réalise $\lfloor n/2 \rfloor$ itérations, on a 1 test conditionnel (celui du tant que) qui utilise 3 opérations, 1 test conditionnel (celui du si) qui utilise aussi 3 opérations → 6 opérations.

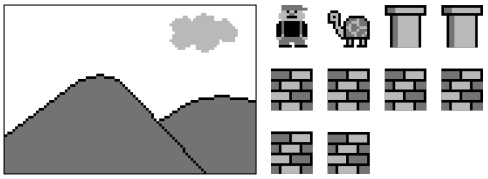
   Par conséquent, on a en tout $\lfloor 6n/2 \rfloor + 2 = 3n+2$ opérations élémentaires. La complexité de cet algorithme est donc linéaire, et la fonction `palindrome` est en $\mathcal{O}(n)$.

Exercice 4 (codage d'informations)

Voici un écran du jeu vidéo *Super Plombiers Italiens* :



Cet écran consiste en un fond de 96 pixels de largeur et 64 pixels de hauteur, sur lequel sont superposés des *sprites*, c'est-à-dire des images carrées de 16×16 pixels qui représentent les objets du jeu, dans ce cas le plombier Marco, son ennemi la tortue, les deux tuyaux et 6 groupes de briques :



Cette version du jeu n'utilise que quatre couleurs (noir, gris foncé, gris clair et une couleur transparente, qui permet notamment de voir le fond derrière les personnages) pour les graphismes.

1. Combien de bits sont-ils nécessaires pour représenter le fond d'écran ? Combien d'octets ?

   Solution : Comme il n'y a que $4 = 2^2$ couleurs, chaque pixel peut être représente par 2 bits. Au total, on a donc $96 \times 64 \times 2 = 12288$ bits pour le fond, soit $12288/8 = 1536$ octets.

2. Combien de bits sont-ils nécessaires pour représenter chaque sprite ? Combien d'octets ?

   Solution : Les sprites utilisent également 2 bits par pixel, donc un total de $16 \times 16 \times 2 = 512$ bits, soit $512/8 = 64$ octets.

Pour chaque écran du jeu, l'ordinateur a besoin de stocker les images du fond et de chaque sprite, mais ce n'est pas nécessaire de maintenir plusieurs copie de l'image de chaque sprite dans la mémoire s'il apparaît plusieurs fois : on peut tout simplement garder trace des coordonnés sur l'écran de chaque occurrence d'un sprite (plus spécifiquement, des cordonnées du point en bas à gauche de chaque sprite). Dans l'écran d'exemple, des briques identiques se trouvent aux coordonnés $(0, 0)$, $(16, 0)$, $(32, 0)$, $(48, 0)$, $(64, 0)$, et $(80, 0)$; deux tuyaux identiques se trouvent aux coordonnés $(0, 16)$ et $(64, 16)$. Marco se trouve aux coordonnées $(0, 32)$ et la tortue aux coordonnées $(32, 16)$.

3. Combien de bits sont-ils nécessaires pour représenter les coordonnés d'un sprite ? Combien d'octets ?

   Solution : L'écran consiste en 96 pixels en largeur. La plus petite puissance de 2 supérieure où égale à 96 est $128 = 2^7$. Il faut donc 7 bits pour représenter la coordonnée x d'un sprite, soit 1 octet. Il y a 64 pixels en hauteur, et comme $64 = 2^6$, il faut 6 bits pour la coordonnée y, soit 1 octet. En total on a donc besoin de 2 octets pour représenter les coordonnées d'un sprite.
   En alternative, on peut compter $7 + 6 = 13$ bits en total, ce qui correspond toujours à 2 octets.

Donc, au delà du fond et d'un exemplaire de chaque image de sprite, on n'a besoin de stocker qu'une séquence de données de la forme (*numéro de sprite*, *coordonnés du sprite*) pour pouvoir reconstruire l'écran. Supposons que le jeu *Super Plombiers Italiens* comprenne un total de 256 sprites différents, qui représentent les protagonistes, les ennemis, les objets bonus, etc.

4. Combien d'octets sont-ils nécessaires pour représenter l'écran d'exemple (y compris le fond et les sprites) ?

   Solution : On a déjà calculé qu'il faut 1536 octets pour le fond et 64 octets pour chacun des 4 types de sprite (Marco, la tortue, le tuyaux, les briques), donc $1536 + 4 \times 64 = 1792$. En plus, il faut stocker les coordonnées des 10 occurrences des sprites avec le type de sprite :
   — $(\text{Marco}, 0, 32)$
   — $(\text{tortue}, 32, 16)$
   — $(\text{tuyau}, 0, 16)$, $(\text{tuyau}, 64, 16)$
   — $(\text{briques}, 0, 0)$, $(\text{briques}, 16, 0)$, $(\text{briques}, 32, 0)$, $(\text{briques}, 48, 0)$, $(\text{briques}, 64, 0)$, $(\text{briques}, 80, 0)$
   Puisque il y a $256 = 2^8$ types de sprites, un entier de 8 bit, soit 1 octet, est nécessaire les identifier, suivi par 2 octets pour les coordonnées d'une occurrence du sprite. Cela nous donne $10 \times (1 + 2) = 30$ octets. En total il faut donc $1792 + 30 = 1822$ octets pour représenter l'écran de l'exemple.

5. De combien de mémoire (en kilo-octets) l'ordinateur doit-il disposer au minimum si chaque écran de jeu peut contenir un maximum de 16 sprites ?

Solution : Si les 16 sprites sur l'écran sont tous différents, il faudra $64 \times 16 = 1024$ octets pour stoker les images des sprites, plus $(1 + 2) \times 16 = 48$ octets pour le type et les coordonnées de chaque sprite sur l'écran. Il faut aussi compter le fond d'écran de 1536 octets. En total, cela donne $1024 + 48 + 1536 = 2608$ octets, soit $2,608$ kilooctets.