IN 101 - TD 02

Solution

Attention : désormais l'implémentation des exercices sera à faire en fin de TD, après que nous ayons examiné toutes les questions de réflexion et de structuration des algorithmes. Laissez le clavier...

Dans les exercices, lorsqu'il vous est demandé d'écrire une fonction puis de la tester, implicitement vous devrez écrire un `main` qui permet d'appeler et tester cette fonction. Vous pourrez récupérer les entrées nécessaires pour tester la fonction soit par les arguments de la ligne de commande, soit en utilisant `scanf`.
Pour rappel, la conversion chaîne → entier est faite avec la fonction `atoi`.

1 Division

≪Écrivez une fonction qui calcule et affiche le quotient de 2 entiers pour un nombre de décimales donné, sans 0 non significatifs en fin d'affichage. ≫

Q 1.1. Quels sont les domaines d'entrée et de sortie de cette fonction ?

Solution

Cette fonction prend en arguments 3 entiers. Comme elle doit **afficher**, elle ne retourne rien, donc a **void** pour type de retour.

Q 1.2. Que pensez-vous de l'utilisation de l'opérateur `/` pour le but recherché ?

Solution

L'opérateur `/` appliqué à 2 entiers calcule la division **entière** et omet donc les décimales. Même appliqué à des flottants, l'utilisation de `/` ne permet en aucun cas de contrôler le nombre de décimales demandé.

Il faut donc programmer la division selon la méthode apprise ...à l'école primaire, avec des suites de divisions entières et de reports de restes.

Q 1.3. Quelles sont les parties qui composent un affichage résultat ?

Solution

Il a être composé d'une partie entière suivie, **éventuellement**, d'un point et d'une partie décimale. En effet, si la division tombe juste, il n'y aura pas besoin de partie décimale.

Q 1.4. Un programme devant réagir correctement à toutes les configurations de données qui lui sont soumises, quelle doit être la première vérification de votre programme ?

Solution

Il faut vérifier par un que le diviseur n'est pas nul. Si c'est le cas, on affichera une erreur.

Q 1.5. Comment allez-vous calculer la partie entière du quotient ? Donnez la forme générale de l'algorithme.

Solution

La partie entière du quotient étant justement ≪entière ≫, elle sera calculée avec l'opérateur / de `C` appliqué à nos 2 entiers (dividende et diviseur). La forme de notre algorithme est donc :

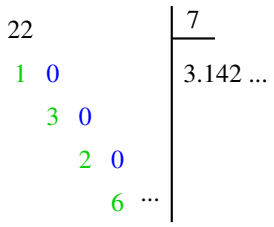
```
div (dividende, diviseur, nb_decim) =
  Si diviseur == 0 alors erreur et quitter
  Afficher (dividende / diviseur)
  S'occuper de l'éventuelle partie décimale
```

Q 1.6. Pour la partie décimale, comment sait-on s'il y en a une à afficher et comment obtient-on les chiffres ?

Solution

Il va y avoir une partie décimale à afficher si le reste de la division entière n'est pas nul. On aura donc un point à afficher uniquement dans ce cas.

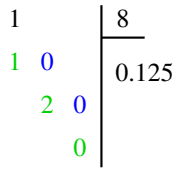
Pour obtenir les chiffres, nous allons répéter le processus habituel de la division telle que l'on l'a apprise à l'école primaire. Pour obtenir le prochain chiffre, on multiplie le reste par 10 (ce qui revient à ≪abaisser un zéro ≫) et on divise de nouveau.



Q 1.7. Comme l'on souhaite ne pas voir s'afficher de 0 inutiles en queue de résultat, à quelle condition le calcul (et donc de l'affichage) des décimales doit-il continuer ?

Solution

Puisque l'on veut afficher un certain nombre, `nb_decimales`, de décimales, tant qu'on n'aura pas atteint ce nombre, il faudra continuer. Mais puisque l'on veut aussi ne pas afficher de zéros inutiles, il faudra s'assurer **aussi** que le reste de la division n'est pas nul. Donc, on continuera à calculer tant que ces 2 conditions sont satisfaites.



Q 1.8. Donnez la forme de l'algorithme de la fonction `division`.

Solution

```
div (dividende, diviseur, nb_decim) =
  Si diviseur == 0 alors erreur et quitter
  Afficher (dividende / diviseur)
  reste <- dividende % diviseur
  Si reste != 0 afficher un '.'
  Tant que reste != 0 et nb_decim > 0
    dividende <- reste * 10
    quotient <- dividende / diviseur
    reste <- dividende % diviseur
    Afficher quotient
    nb_decim <- nb_decim - 1
```

Q 1.9. Que se passe-t-il si le dividende ou le diviseur sont négatifs ? Si besoin, rectifiez l'algorithme.

Solution

Si **l'un des deux seulement** est négatif, les divisions successives vont donner des chiffres négatifs. Donc nous allons afficher des chiffres avec un `'-'` devant à chaque fois. Passons sur le fait qu'en plus, le modulo de nombres négatifs qui n'a pas la même signification selon les langages de programmation !

Si **les deux** sont négatifs tout se passe bien car le quotient (produit) de deux nombres négatifs est positif.

Ainsi, si l'un des deux est négatifs, il faut afficher un signe `'-'` au début, et changer le signe de celui qui est négatif.

```
div (dividende, diviseur, nb_decim) =
  Si dividende == 0 alors erreur et quitter
  Si dividende et diviseur de signes différents
    Afficher '-'
    Si dividende < 0 alors dividende <- -dividende
    Sinon diviseur <- -diviseur
  Afficher (dividende / diviseur)
  ...
```

2 Tri

≪Écrivez une fonction vérifiant si un tableau d'entiers est trié. ≫

Q 2.1. Quels sont les domaines des entrées et des sorties ?

Solution

Entrées : La fonction doit prendre en argument le tableau d'entiers. Comme la taille d'un tableau ne fait pas partie du tableau, il faut qu'elle prenne également sa taille. La taille ne pouvant pas être négative, ce sera un entier non signé (`unsigned int`).

Sorties : La fonction devant ≪vérifier ≫, elle doit répondre par ≪oui ≫ ou ≪non ≫, ≪vrai ≫ ou ≪faux ≫. La valeur retournée est donc une valeur de **vérité**, c'est donc un **booléen**.

Q 2.2. Que signifie ≪est trié ≫ ? Quel va être le choix qui devra être fait pour l'algorithme ?

Solution

L'énoncé ne spécifie **aucun ordre** de tri : ni croissant ni décroissant. C'est une spécification très floue et mal posée (c'est fait exprès).

Si un tableau est trié en ordre croissant **ou** trié en ordre décroissant alors **il est trié**. Donc, on va vérifier s'il l'est dans un des **deux** ≪sens ≫.

Et si le tableau **ne contient pas d'éléments**, est-il trié ? Il y a donc un **cas particulier** à gérer. Il n'y a pas de réponse ferme, on peut décider qu'il est trié en partant du principe que si l'on trie un tableau vide, on obtient le **même** tableau, donc il était déjà trié.

Q 2.3. Définissez formellement le critère ≪être trié ≫ pour un tableau. Quel est l'impact de ≪strict ≫ ou ≪ou égal ≫ ?

Solution

Un tableau t de taille l est trié selon la relation $\preccurlyeq$ si $\forall i\, j \in [0; l[,\ i < j \Rightarrow t[i] \preccurlyeq t[j]$.

Que la relation (d'ordre) soit ≪inférieur ≫ ou ≪supérieur ≫, il reste le choix de ≪strict ≫ ou ≪égal ≫ (même si $<$ n'est pas une relation d'ordre puisqu'elle n'est pas réflexive, on parle plutôt d'ordre *strict*).

Si le choix est ≪strict ≫ alors un tableau contenant **plusieurs éléments identiques consécutifs** (i.e. $t[i] = t[j]$) ne sera **jamais trié** puisque l'on n'aura jamais $t[i] < t[j]$ (ou $>$).

Dans la suite de cette correction, le choix de ≪**égal** ≫ est décidé.

Q 2.4. En vous appuyant sur la définition formelle d'un tri vue en question Q2.3, proposez la structure d'un algorithme répondant à la question du tri.

Solution

Puisque le tableau peut être trié de manière croissante ou décroissante il faut envisager les 2 cas. En regardant la première et la dernière case, on déduit la façon dont il peut être trié.

- Si $t[0] = t[l-1]$ alors s'il est trié, peu importe la manière c'est que tous les éléments sont égaux. On gérera ça avec l'un des cas ci-dessous.
- Si $t[0] < t[l-1]$ alors s'il est trié, c'est de manière croissante : il faudra vérifier avec $\geq$.
- Sinon, s'il est trié, c'est de manière décroissante : il faudra vérifier avec $\geq$.

On va balayer le reste du tableau (de l'indice **1** à $l-1$) en regardant si $t[i] \leq t[i-1]$ ou $\geq$ en fonction de l'ordre déterminé précédemment. Dans les faits, on va inverser les deux tests afin de retourner faux si l'on détecte que deux éléments consécutifs ne sont pas triés. Cela évite d'écrire un algorithme de la forme :

```
...
si t[i-1] <= t[i] rien faire        (respectivement >= dans l'autre cas)
sinon retourner faux
...
```

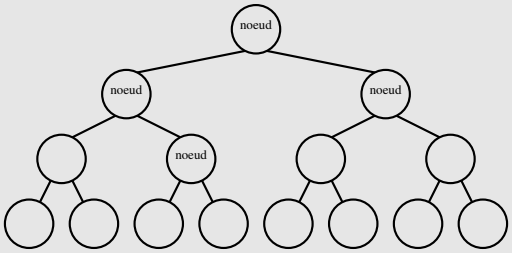
Si le tableau n'a qu'une seule case, il est forcément trié, donc nous pouvons le gérer avec le cas d'un tableau de taille nulle.

```
sorted (t, l) =
  si l < 2 retourner vrai
  test_croiss = t[0] <= t[l-1]
  pour i de 1 à l - 1 inclus,
    si test_croiss = vrai alors
      si t[i-1] > t[i] retourner faux
    sinon
      si t[i-1] < t[i] retourner faux
  retourner vrai
```

3 Arbre

≪Écrivez une fonction permettant d'afficher un arbre binaire complet de manière textuelle. ≫

En informatique, un arbre est une structure où des *nœuds* sont reliés à d'autres nœuds (*fils*) de manière arborescente (cf. IN103). Un arbre est *binaire* si chaque nœuds **a au plus 2** fils. Un arbre est *complet* si tous ses nœuds **terminaux** ≪sont au même niveau ≫. Un schéma valant mieux qu'un long discours :



On souhaite afficher uniquement les nœuds d'un tel arbre avec des espaces et le caractère `'*'` en fonction d'une hauteur h donnée en argument de la fonction. Les noeuds à la base de l'arbre sont séparés par **1 seul espace**.

Exemples :

— Pour une hauteur 2 :

```
 *

* *
```

— Pour une hauteur 3 :

```
   *

 *   *

* * * *
```

— Pour une hauteur 4 :

```
     *

   *       *

 *   *   *   *

* * * * * * * *
```

Q 3.1. Quels sont les domaines des entrées et des sorties ?

Solution

Entrées : La fonction doit prendre en argument la hauteur de l'arbre à afficher : c'est donc un entier. Puisque la hauteur ne peut pas être négative, autant prendre un **unsigned int**.

Sorties : La fonction doit juste afficher, donc elle n'a aucune information à retourner. Le type de ≪rien≫ est **void** en `C`. On parle parfois de ≪procédure≫ pour dénoter une fonction retournant ≪rien≫.

Cette fonction, au lieu de retourner une valeur, a ≪un effet≫ : le résultat de son exécution se manifeste autrement que par une valeur retournée. On parle ≪d'effet de bord≫.

Q 3.2. De quoi est composé l'affichage ? De quelles relations avez-vous besoin pour faire cet affichage ?

Solution

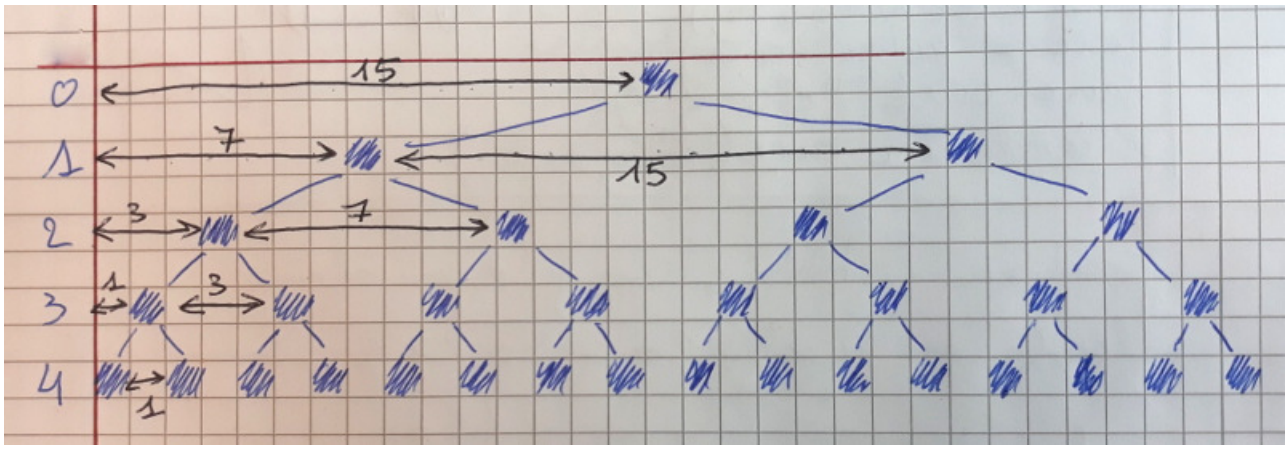
L'affichage est composé de lignes de texte. Les lignes séparant les ≪étages≫ sont vides. Les lignes non vides sont composées d'espaces en début, puis de `'*'` séparés par des espaces.

Il nous faut donc trouver les relations entre ≪l'étage≫ de l'arbre que l'on affiche et :
— le nombre d'espaces en début de ligne (appelons-le `indent`),
— le nombre de `'*'` à afficher ,
— le nombre d'espaces entre les `'*'` (appelons-le `spacing`).

Q 3.3. Donnez l'expression formelle de ces relations.

Solution

Faisons un petit dessin...



— On remarque que le nombre d'espaces entre 2 `'*'` (ie. `spacing`) est une puissance de 2 moins 1. Laquelle ?
  — Étage 0 : sans importance, il n'y a qu'une seule `'*'`.
  — Étage 1 : 15 c'est en fait $2^{\text{hauteur de l'arbre}-1} - 1 = 2^{5-1} - 1 = 2^4 - 1$.
  — Étage 2 : 7 c'est en fait $2^{\text{hauteur de l'arbre}-2} - 1 = 2^{5-2} - 1 = 2^3 - 1$.
  — Étage 3 : 3 c'est en fait $2^{\text{hauteur de l'arbre}-3} - 1 = 2^{5-3} - 1 = 2^2 - 1$.
  — On en déduit que pour l'étage l c'est $2^{\text{hauteur de l'arbre}-l} - 1$

— On remarque que le nombre d'espaces en début de ligne (ie. `indent`) est égal à $\frac{1}{2}$ de `spacing`.
— On remarque que le nombre de `'*'` à afficher est $2^{\text{étage de l'arbre}}$.

Q 3.4. Donnez la structure grossière de l'algorithme.

Solution

Pour chaque étage de 0 à la limite passée en argument de la fonction, on va calculer `spacing` et `indent`, puis on va imprimer `indent` espaces, puis la première `'*'`, ensuite on va compléter la ligne en affichant `spacing` espaces suivi d'une étoile pour les $2^{\text{étage de l'arbre}} - 1$ `'*'` restants et on termine par 2 sauts de ligne.

Pour calculer 2^n, on pourrait être tenté de faire une fonction particulière. Toutefois, nous pouvons obtenir ce calcul de manière très simple : 2^n c'est 1 décalé n fois vers la gauche, ce qui s'écrit en C : `1 << n`.

En effet, les entiers sont codés en base 2. Faire un décalage de 1 à gauche n fois est bien calculer le nombre 2^n. C'est tout comme en base 10 où décaler 1 à gauche n fois c'est calculer 10^n.

On remarque que l'on doit plusieurs fois afficher un nombre d'espace donné. Autant faire une fonction `print_spaces` qui s'en charge et prend en argument le nombre d'espaces à imprimer.

4 Implémentation

Écrivez en `C` les algorithmes esquissés dans les exercices 1, 2 et 3. Pour l'exercice 2, un squelette de programme vous est donné (`sorted_skel.c`) comportant de nombreux cas de test (fastidieux à écrire).

Solution

4.1 `division`

```c
#include <stdio.h>
#include <stdlib.h>

void division (int dividende, int diviseur, int nb_decim) {
  int reste, quotient ;

  /* Ne pas oublier le cas de la division par 0. */
  if (diviseur == 0) {
    printf ("Division by 0.\n") ;
    return ;
    }

  if ((dividende > 0) != (diviseur > 0)) {
    printf ("-") ;
    if (dividende < 0) dividende = - dividende ;
    else diviseur = - diviseur ;
  }

  /* Calcul de la partie entière. */
  quotient = dividende / diviseur ;
  reste = dividende % diviseur ;
  printf ("%d", quotient) ;
  /* Point uniquement si partie décimale non vide. */
  if (reste != 0) printf (".") ;
```

```
  /* Itération pour la partie décimale. */
  while ((nb_decim > 0) && (reste != 0)) {
    dividende = reste * 10 ;  /* On "abaisse" toujours un 0. */
    quotient = dividende / diviseur ;
    reste = dividende % diviseur ;
    printf ("%d", quotient) ;
    nb_decim-- ;
  }

  if (reste == 0) printf ("\nRésultat exact\n.") ;
  else printf ("\nRésultat approché (reste %d)\n.", reste) ;
}


int main (int argc, char *argv[])
{
  if (argc != 4) {
    printf ("Erreur. Mauvais nombre d'arguments.\n") ;
    return 1 ;
  }

  division (atoi (argv[1]), atoi (argv[2]), atoi (argv[3])) ;
  return 0 ;
}
```

4.2 sorted

```
#include <stdbool.h>
#include <stdio.h>

bool sorted (int *t, unsigned int l) {
  /* Consider that the empty array is sorted. An array with one 1
     cell is also always sorted. */
  if (l < 2) return true ;

  /* Check which way to check the sorting. true means increasing order. */
  bool verif_increase = (t[0] <= t[l - 1]) ;
  for (int i = 1; i < l; i++) {
    if (verif_increase) {
      /* Invert the test to detect wrong order of the elements
         according to the selected order hence directly exit. */
      if (t[i - 1] > t[i]) return false ;
    }
    else {
      if (t[i - 1] < t[i]) return false ;
    }
  }

  /* If here, then we never exited on a wrong order of the elements. Hence
     the array is really sorted. */
  return true ;
}
```

4.3 print_tree

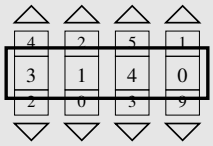
```
#include <stdio.h>
#include <stdlib.h>

void print_spaces (unsigned int number) {
  for (int i = 0; i < number; i++) printf (" ") ;
}


void print_tree (unsigned int height) {
  for (int line = 0; line < height; line++) {
    /* Left shift computes a power of 2. Saves a power function. */
```

```
    int spacing = (1 << (height - line)) - 1 ;
    int indent = spacing / 2 ;
    int nb_nodes = 1 << line ;
    /* Print spaces starting the line. */
    print_spaces (indent) ;
    /* Print the first star apart to avoid trailing white spaces. */
    printf ("*") ;
    for (int i = 1; i < nb_nodes; i++) {
      print_spaces (spacing) ;
      printf ("*") ;
    }
    /* 2 ends of line. */
    printf ("\n\n") ;
  }
}
```

S'il vous reste du temps ou pour continuer après la séance.

5 Cadenas à roulette

La solution d'un examen se trouve enfermée dans un coffre dont l'ouverture est protégée par un cadenas numérique à roulettes. Un tel cadenas est composé d'un certain nombre de roulettes chacune graduée des chiffres de 0 à 9 dans un ordre consécutif. Il est possible de faire défiler chaque roulette soit vers le haut soit vers le bas afin de changer les chiffres composant l'état actuel du cadenas. Lorsque le code affiché correspond à celui prédéfini dans le cadenas, ce dernier s'ouvre.



Vous connaissez le code de déverrouillage et le code actuellement affiché par le cadenas. Sauf que le cadenas est tellement rouillé que chaque tour de molette supplémentaire risque de le gripper définitivement. Il faut donc trouver le nombre minimum de rotations à effectuer sur les molette pour ouvrir le cadenas. C'est ce que devra calculer et afficher votre programme.

Q 5.1. Comment peut-on représenter simplement le code de déverrouillage et le code actuellement affiché sur le cadenas ?

Solution

Il suffit de les représenter par des entiers. Sachant que les codes ne représentent que des valeurs positives, nous pourrions gagner à utiliser des `unsigned int`, ce qui nous éviterait structurellement un test pour vérifier que l'on n'a pas reçu des valeurs négatives en entrée. Mais passons.

Nous allons bien entendu vouloir structurer notre programme en une fonction effectuant le calcul et un `main` se chargeant de l'administratif autour.

Q 5.2. Quels sont les domaines d'entrée et de sortie de la future fonction effectuant le calcul ?

Solution

Cette fonction prendra deux (unsigned) `int` en entrée (le code de déverrouillage et l'état initial du cadenas) et retournera un entier, (non signé, encore une fois, si l'on souhaite être puriste).

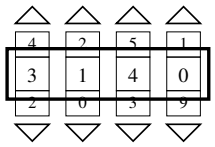
Q 5.3. Proposez un algorithme permettant de calculer ce nombre minimal de rotations.

Solution

Raisonnons d'abord sur une seule roulette du cadenas, donc un seul chiffre. On se rend compte qu'il y a 3 cas de figure pour atteindre le chiffre visé à partir du chiffre actuel :

— il est plus rapide de tourner la molette vers le haut (diminuer le chiffre sur le schéma proposé),
— il est plus rapide de tourner la molette vers le bas (augmenter le chiffre sur le schéma proposé),
— les deux sens se valent.

Cas	(c_1)	(c_2)	(c_3)	(c_4)	(c_5)	(c_6)
Code visé (c)	0	4	2	9	5	0
Chiffre actuel (a)	**4**	**0**	**9**	**2**	**0**	**5**
Sens	↑	↓	↓	↑	↕	↕
Nb. tours	4	4	3	3	5	5



Soit c le chiffre visé et a le chiffre actuel. Examinons les différentes configurations possibles.

Dans le cas (c_1) on voit qu'il est plus rapide d'aller de $a = 4$ vers $e = 0$ en diminuant a. Donc le nombre minimal de tours les $a - c$. Dans le cas (c_2), il est plus rapide d'augmenter a d'un nombre de tours égal à $c - a$. Donc, dans ces deux cas le nombre minimum est $|c - a|$.

Dans le cas (c_3) il est plus rapide d'augmenter c en 3 tours, pour le faire en quelque sorte arriver à 12. Donc le nombre de tours minimum est alors $10 - (c - a)$. Dans le cas (c_4), c'est en $10 - (a - c)$. Donc, dans ces deux cas le nombre minimum est $10 - |c - a|$.

Bref, le nombre de tours minimum est donc $min(|c - a|, 10 - |c - a|)$. Et dans les cas (c_5) et (c_6), il se trouve bien que $|c - a| = 10 - |c - a|$ donc le min ne changera rien.

Nous savons maintenant comment calculer le nombre minimal de tours pour un chiffre de notre cadenas. Ce cadenas en comporte plusieurs, il nous suffit donc d'itérer ce processus sur chacun des chiffres des nombres représentant l'état actuel du cadenas et le code visé. Pour récupérer les chiffres d'un nombre, il suffit de prendre le modulo 10 du nombre, ce qui nous donne le chiffre le plus à droite, puis de diviser ce nombre par 10 pour se décaler vers la gauche. On va répéter ce processus tant que l'état actuel du cadenas et le code visé ne seront pas réduits à 0.

On obtient donc l'algorithme suivant :

```
unlock (actuel, code) =
  nb_tours = 0
  tant que etat <> 0 ou code <> 0
    chiffre_code = code % 10
    chiffre_actuel = actuel % 10
    nb_tours = nb_tours + min (|chiffre_actuel - chiffre_code|,
                               10 - |chiffre_actuel - chiffre_code|)
    chiffre_code = chiffre_code / 10
    chiffre_actuel = chiffre_actuel / 10
  retourner nb_tours
```

On remarque au passage qu'il n'est pas nécessaire de calculer 2 fois `|chiffre_actuel - chiffre_code|`.

```c
#include <stdio.h>
#include <stdlib.h>

int min (int a, int b) { return a < b ? a : b ; }


int unlock (int current, int code) {
  int nb_turns = 0 ;

  while (current != 0 || code != 0) {
    int curr_d = current % 10 ;
    int code_d = code % 10 ;
    int diff = abs (curr_d - code_d) ;
    nb_turns += min (diff, 10 - diff) ;
    current /= 10 ;
    code /= 10 ;
  }

  return nb_turns ;
}


int main (int argc, char *argv[]) {
  if (argc != 3) {
    printf ("Error. Usage: <unlock code> <current state>.\n") ;
    return 1 ;
  }

  int current = atoi (argv[2]) ;
  int code = atoi (argv[1]) ;
  printf ("Nb turns: %d\n", unlock (current, code)) ;
  return 0 ;
}
```

6 Chaîne d'heure

≪Écrivez une fonction qui **affiche** la chaîne de caractères représentant une heure telle que l'on la dit à l'oral en français. ≫

Le cours débute à ≪quinze heures moins le quart ≫. Le TD commence à ≪seize heures ≫. On change de jour à ≪minuit ≫. À ≪une heure et demi ≫, il fait nuit, pareil à ≪deux heures moins vingt cinq ≫.

On souhaite ainsi obtenir la chaîne de caractères représentant une heures ≪à l'ancienne ≫, en ne disant pas ≪quarante cinq ≫ ou ≪trente ≫. On vous épargne la gestion des tirets dans les nombres composés (≪quarante cinq ≫ au lieu de ≪quarante-cinq ≫ ira très bien).

Le but n'est pas de faire 2 grosses suites de `if/else if`, une pour les 23 heures et une pour les 59 minutes ! Ceci n'aurait aucun intérêt algorithmique et serait fort fastidieux.

La solution de cet exercice n'est pas donnée dans ce PDF car le code source est un peu long. Vous pourrez néanmoins consulter ce dernier puisqu'il figure dans l'archive solution qui vous est fournie.

Q 6.1. Quels sont les cas particuliers des heures ?

Solution

Il y a 0h qui se traduit en ≪minuit ≫ et 12h qui se traduit en ≪midi ≫.

Q 6.2. Quels sont les cas particuliers des minutes ?

Solution

Il y a 15 qui se traduit en ≪le quart ≫, 30 en ≪et demi ≫, 45 en ≪moins le quart ≫.

On remarque ensuite que les minutes sont globalement séparées en 2 : celles avant 30 et celles après. Dans le premier cas on les traduit en une certaine chaîne, dans le second, on met ≪moins ≫ devant et on traduit comme des minutes entre 0 et 30 (ce dernier étant exclu).

Q 6.3. Réfléchissez à ce que vous pouvez factoriser dans l'algorithme pour réutiliser le même traitement dans différents cas. Attention, il traîne encore quelques cas particuliers de la langue française non mentionnés en Q6.1.

Q 6.4. Programmez la fonction `string_of_time` demandée. Attention à ne pas rajouter d'espaces inutiles. Vous pouvez tester votre fonction avec (au moins) les cas suivants :

```
string_of_time (12, 0)   → "midi"
string_of_time (00, 0)   → "minuit"
string_of_time (00, 15)  → "minuit et quart"
string_of_time (00, 30)  → "minuit et demi"
string_of_time (00, 42)  → "une heure moins dix huit"
string_of_time (12, 0)   → "midi"
string_of_time (12, 45)  → "treize heures moins le quart"
string_of_time (21, 21)  → "vingt et une heures vingt et une"
string_of_time (7, 30)   → "sept heures et demi"
string_of_time (1, 25)   → "une heure vingt cinq"
string_of_time (12, 20)  → "midi vingt"
string_of_time (17, 35)  → "dix huit heures moins vingt cinq"
string_of_time (23, 59)  → "minuit moins une"
string_of_time (23, 25)  → "vingt trois heures vingt cinq"
```