ENSTA Paris
2024-2025

Résolution de problèmes algorithmiques – IN103

Alexandre Chapoutot et Marine Saint-Mézard

Feuille d'exercices 2

Objectif(s)

★ Prendre en main les structures de données linéaires : liste, pile, file et ensemble.

PRÉPARATION

Cette première partie permet de préparer votre environnement de travail afin de pouvoir utiliser facilement la bibliothèque logicielle **libin103** spécialement développée pour cet enseignement.

1. A la racine de votre compte, créez un répertoire nommé `Library`, puis placez vous dans ce répertoire.

   ```
   mkdir ~/Library; cd ~/Library
   ```

2. Téléchargez l'archive `libin103-1.4.2.tar.gz` sur le site du cours

   ```
   wget
   https://perso.ensta-paris.fr/~chapoutot/teaching/in103/practical-work/libin103-1.4.2.tar.gz
   ```

3. Désarchivez l'archive

   ```
   tar -xvzf libin103-1.4.2.tar.gz
   ```

4. Allez dans le répertoire `libin103-1.4` et compilez la bibliothèque. Quelle commande faut-il utiliser ?
   — Il faut utiliser la commande `make`. A la fin de la compilation vérifier la présence du fichier `libin103.a` dans le répertoire `source`.
   — Également, vous pouvez exécuter la commande `make check` pour compiler et exécuter les programmes de tests.

Matériel pour le TP

Récupérez l'archive associé à cette séance de TP à l'adresse :

```
https://perso.ensta-paris.fr/~chapoutot/teaching/in103/practical-work/in103-td2.tar.gz
```

EXERCICES

Exercice 1 – Prise en main des listes chaînées

L'objectif de cet exercice est de prendre en main la partie de la bibliothèque qui concerne les listes chaînées.

Fonctions utiles pour cet exercice

- `integer_list_init`
- `integer_list_destroy`
- `integer_list_ins_next`
- `integer_list_rem_next`
- `integer_list_size`
- `integer_list_head`
- `integer_list_tail`
- `integer_list_next`
- `integer_list_data`

Remarque : Les données de cet exercice sont dans le répertoire `exo1`.

Question 1

Écrivez le fichier `Makefile` pour compiler le programme `list-printer.x` à partir du fichier source `list-printer.c`. Pensez à ajouter une règle `clean` pour supprimer les fichiers générés lors de la compilation.

Solution:

```
CC=gcc
CFLGAS=-Wall -Werror -I$(HOME)/Library/libin103-1.4.2/include
LDFLAGS=-L$(HOME)/Library/libin103-1.4.2/source -lin103

all: list-printer.x

list-printer.x: list-printer.o
	$(CC) $(CFLGAS) -o $@ $< $(LDFLAGS)

%.o : %.c
	$(CC) $(CFLGAS) -c $<

clean:
	rm -f *~ *.o

realclean: clean
	rm -f *.x
```

Explication(s):

- La principale difficulté est d'utiliser les options du compilateur C pour trouver les fichier `.h` et la bibliothèque `libin103.a`.
- Suivant les instructions d'installation de la bibliothèque `libin103` : l'ajout du répertoire contenant les fichiers `.h` se réalise avec l'option `-I$(HOME)/Library/libin103-1.4/include`
- Suivant les instructions d'installation de la bibliothèque `libin103` : l'ajout de la bibliothèque`libin103.a` se réalise avec l'option `-L$(HOME)/Library/libin103-1.4/source` et l'option `-lin103` qui est le raccourci pour ajouter le fichier `libin103.a`.

Question 2

Écrivez le code associé à la fonction `print_list` dans le fichier `list-printer.c`.

Solution:

```
void print_list (integer_list_t *list) {
  integer_list_elmt_t* elt = integer_list_head(list);
  for (; elt != integer_list_tail(list); elt = integer_list_next(elt)) {
    printf ("%d, ", integer_list_data(elt));
  }
  printf ("%d\n", integer_list_data(integer_list_tail(list)));
}
```

Explication(s):

- Nous avons ici une forme classique d'itérateur pour lequel on récupère le pointeur de tête, on avance de maillon en maillon jusqu'au dernier maillon.
- La mise en œuvre de cet itérateur est facilité par l'API des listes chaînées qui fournit toutes les fonctions utiles.

Exercice 2 – Problème de couverture par ensembles

Les problèmes de couvertures d'ensemble sont des problèmes d'optimisation combinatoires qui apparaissent dans divers domaines comme les problèmes de logistiques, par exemple, placement d'un nombre minimum de centres de distribution maximisant la couverture de population, ou le placement de caméras pour couvrir une zone.

Ces problèmes de couvertures d'ensembles entre dans la classe des problèmes algorithmiques difficiles (classe NP) pour lesquels des solutions approchées peuvent être calculées par des algorithmes gloutons. L'objectif de cet exercice est de mettre en œuvre un tel algorithme.

Fonctions utiles pour cet exercice (en plus de celles sur les listes)

- `integer_set_init`
- `integer_set_destroy`
- `integer_set_insert`
- `integer_set_remove`
- `integer_set_union`
- `integer_set_difference`
- `integer_set_intersection`
- `integer_set_size`

Formulation mathématique du problème est étant donné
— un ensemble U à couvrir ;
— un ensemble d'ensembles S ;
on cherche le plus petit $S' \subseteq S$ tel que $\forall u \in U, u \in S'$.

Par exemple, si $U = \{1,2,4,6,8,9\}$ et $S = \{\{1\},\{2\},\{2,4\},\{4,6\},\{1,6,8\},\{8\},\{9\}\}$ on a :
— $S_1 = \{\{1\},\{2\},\{4,6\},\{8\},\{9\}\}$ est une couverture de U de taille 5 ;
— $S_2 = \{\{2,4\},\{4,6\},\{1,6,8\},\{9\}\}$ est aussi une couverture de U mais de taille 4 ;
— $S_3 = \{\{4,6\},\{1,6,8\},\{9\}\}$ n'est pas une couverture de U car $2 \in U$ mais $2 \notin S_3$.

En s'appuyant sur l'exemple ci-dessus, nous allons développer un algorithme glouton qui trouve une couverture d'ensembles mais qui en général ne sera pas la plus petite solution mais une bonne approximation.

Remarque : Les données de cet exercice sont dans le répertoire `exo5`.

Question 1

Développer une idée d'algorithme glouton pour calculer une couverture d'ensemble.

Solution:
Explication(s):

- ✿ **Rappel à faire aux étudiants** : Nous appliquons une approche gloutonne, c'est-à-dire, une approche itérative qui à chaque itération considère la solution optimale locale afin de calculer un optimum (ou une approximation de l'optimum) global.
- ✿ Dans notre problème de couverture, l'approche gloutonne consiste à chaque itération de l'algorithme à choisir l'ensemble S_i de S qui maximise le nombre d'éléments couverts de U. Puis on répète le processus en considérant l'ensemble U privé des éléments déjà couverts jusqu'à ce que U soit vide.

Question 2

Quelle structure de donnée peut être utilisée pour représenter l'ensemble d'ensembles S ?

Solution:
Un tableau d'ensembles.

Question 3

Écrivez une fonction `glouton` dont le prototype est

```
integer_list_t glouton(integer_set_t* S, int size, integer_set_t* U)
```

— Le type de retour `integer_list_t` représente une liste contenant les indices du tableau S des ensembles qui couvrent U ;
— Les entrées sont :
  — Le tableau S des ensembles ainsi que sa taille `size` ;
  — L'ensemble à couvrir U.

Solution:
Explication(s):

- ✿ L'idée principale est de chercher les ensembles de S qui couvrent le plus les parties de U à couvrir. Pour cela on s'adapte à l'API des ensembles de la bibliothèque qui offre la fonction `integer_set_difference`. Avec cette fonction on cherche à calculer le nombre d'éléments de U qui ne sont pas couvert par $S[i]$ puis on cherche l'indice i associé à l'ensemble qui laisse le plus petit éléments de U non couvert.
- ✿ La recherche de l'indice i nécessite une fonction auxiliaire.
- ✿ La partie pénible de cette fonction est dans la manipulation des ensembles qui fait intervenir pas mal de variables locales.

Plusieurs implémentation sont possibles en fonction des opérations ensemblistes que nous considérons. Deux implémentations sont données dans la suite.

Solution:
(Version 1) **Explication(s):**

- ✿ En utilisant l'opération ensembliste de différence d'ensembles, on considère les éléments de U qui ne sont pas couverts par S_i donc on cherche l'indice du tableau qui minimise cette valeur.

```
int indice_min (int* tab, int size) {
  int pos = 0;
  int min = tab[0];
  for (int i = 1; i < size; i++) {
    if (tab[i] < min){
      pos = i;
      min = tab[i];
    }
  }
  return pos;
}

integer_list_t glouton(integer_set_t* S, int size, integer_set_t* U) {
  int all_size[size];

  integer_list_t resultat;
  integer_list_init(&resultat);

  integer_set_t trouve;
  integer_set_t temp;
  integer_set_t reste;
  integer_set_init(&trouve);
  integer_set_init(&temp);
  integer_set_init(&reste);

  while (integer_set_size(&trouve) < integer_set_size(U)) {
    /* On ne garde que les elements de U qui ne sont pas couverts */
    integer_set_difference(&reste, U, &trouve);
    /* On calcule pour chaque S[i] les elements de U encore \'a couvrir
       qui ne sont pas couverts par S[i] */
    for (int i = 0; i < size; i++) {
      integer_set_difference(&temp, &reste, &S[i]);
      all_size[i] = integer_set_size(&temp);
    }

    /* Le plus petit elements de all_size indique l'element S[i] qui
       couvre le plus les elements de U encore \'a couvrir */
    int pos = indice_min (all_size, size);
    integer_set_union(&temp, &S[pos], &trouve);

    /* Astuce pour copier un ensemble dans un autre */
    integer_set_union(&trouve, &temp, &temp);

    /* On garde l'indice i du tableau S qui correspond localement \'a
       S[i] qui couvre le plus U */
    integer_list_ins_next(&resultat, NULL, pos);
  }
  return resultat;
}
```

Solution:
(Version 2) **Explication(s):**

- ✿ En utilisant l'opération ensembliste d'intersection d'ensembles, on considère les éléments de U qui sont couverts par S_i donc on cherche l'indice du tableau qui maximise cette valeur.

```
int indice_max (int* tab, int size) {
  int pos = 0;
  int max = tab[0];
  for (int i = 1; i < size; i++) {
    if (tab[i] > max){
      pos = i;
      max = tab[i];
    }
  }
  return pos;
}


integer_list_t glouton_v2(integer_set_t* S, int size, integer_set_t* U) {
  int all_size[size];

  integer_list_t resultat;
  integer_list_init(&resultat);

  integer_set_t temp;
  integer_set_t reste;
  integer_set_init(&temp);
  integer_set_init(&reste);

  /* au debut il nous reste tout U \'a trouver */
  integer_set_union(&reste, U, U);

  while (integer_set_size(&reste) > 0) {
    /* On calcule pour chaque S[i] le nombre de nouveaux elements
       qu'il permet de decouvrir */
    for (int i = 0; i < size; i++) {
      integer_set_intersection(&temp, &reste, &S[i]);
      all_size[i] = integer_set_size(&temp);
    }

    /* Le plus grand elements de all_size indique l'element S[i] qui
       couvre le plus les elements de U encore \'a couvrir */
    int pos = indice_max (all_size, size);

    /* on met \'a jour l'ensemble reste */
    integer_set_difference(&temp, &reste, &S[pos]);

    // astuce pour copier une liste dans une autre
    integer_set_union(&reste, &temp, &temp);

    /* On garde l'indice i du tableau S qui correspond localement \'a
       S[i] qui couvre le plus U */
    integer_list_ins_next(&resultat, NULL, pos);
  }
  return resultat;
}
```

Question 4 – Application

La RATP souhaite sécuriser les couloirs d'une station de métro. Dans un objectif de réduction des coûts, elle souhaite utiliser le moins de caméra possible. On considère que les caméras ont une vision à 360° et une vision infinie.

1. Comment peut-on modéliser ce problème pour se ramener au problème de couverture d'ensemble ?
2. Combien de caméra sont nécessaires pour couvrir la station de métro ? et où devons-nous les placer ?

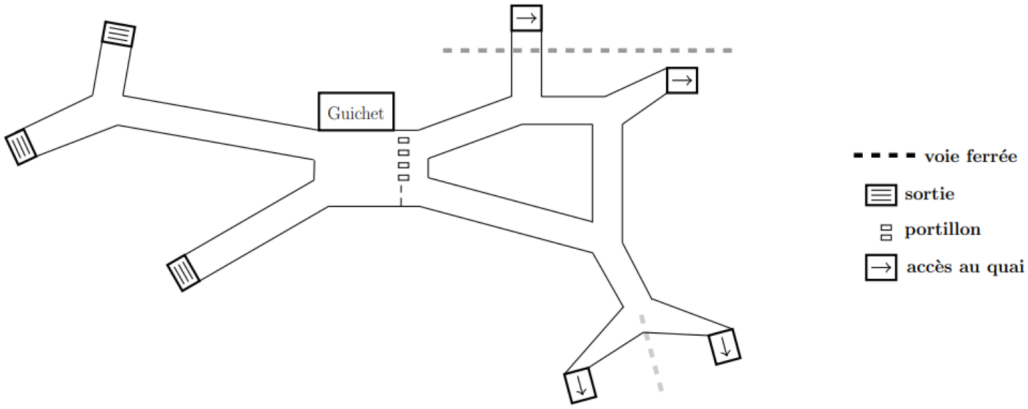
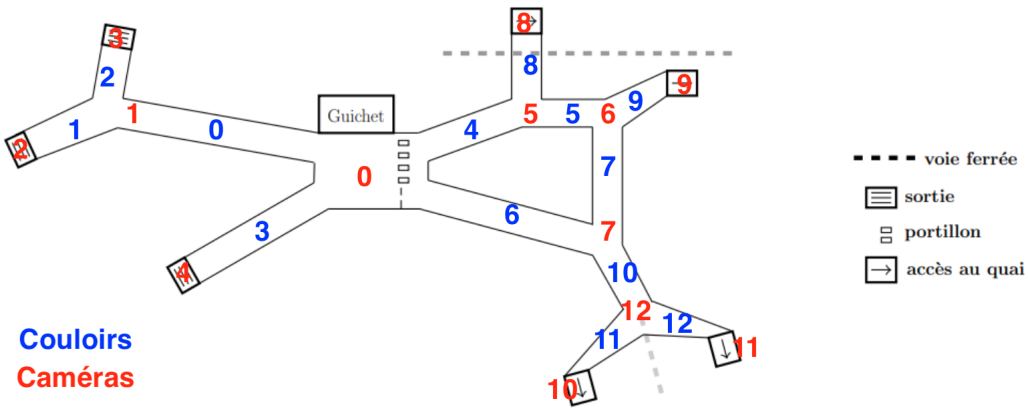


FIGURE 1 – Plan d'une station de métro

Solution:

Explication(s):

- On souhaite se ramener au problème de couverture d'ensembles, on va donc paramétrer le problème pour créer notre tableau d'ensembles.
- On numérote les couloirs de la station de 1 à 13 ce qui constituera notre ensemble U. De manière systématique on considère que les caméras sont placées aux extrémités des couloirs ou aux intersections de couloirs. Par conséquence, on considère 13 emplacements possibles pour les caméras. Pour chaque caméra on crée un ensemble contenant les numéros de couloirs visibles par cette caméra.
- La numérotation utilisée par la solution proposée est donnée dans l'image ci-dessous.
- Une fonction auxiliaire `array_to_set` est définie dans la solution pour faciliter la construction des ensembles.



Solution:

```
void array_to_set(integer_set_t* set, int* array, int size) {
  integer_set_init(set);
  for (int i = 0; i < size; i++) {
    integer_set_insert(set, array[i]);
  }
}

int application() {
  /* Modelisation du probleme: 13 cameras */
  /* Voir la carte avec la numerotation des cameras et couloirs */
  integer_set_t S[13];
  int data[13][4] = {
    {0, 3, 4, 6},
    {0, 1, 2},
    {1},
    {2},
    {3},
    {4, 8, 5},
    {5, 7, 9},
    {7, 10},
    {8},
    {9},
    {11},
    {12},
    {10, 11, 12}
  };
  /* Tailles des sous ensembles de S */
  int size[13] = {4, 3, 1, 1, 1, 3, 3 ,2, 1, 1, 1, 1, 3};

  for (int i = 0; i < 13; i++) {
    array_to_set(&S[i], data[i], size[i]);
  }

  /* Ensemble \‘a couvrir les 13 couloirs de station de metro */
  integer_set_t U;
  integer_set_init(&U);
  for (int i = 0; i < 13; i++) {
    integer_set_insert(&U, i);
  }
  integer_set_t res = glouton(S, 13, &U);
  print_set(&res);

  return 0;
}
```

Exercice 3 – Un peu de réflexion avec les piles et les files

Un grand classique des problèmes algorithmiques données dans les entretiens d'embauche. En supposant que vous n'avez accès qu'à des structures de données de type pile, donnez une mise en œuvre d'une structure de données de type file. Il faut donner une mise en œuvre des fonctions :

- `enqueue`
- `dequeue`

Remarque : Les données de cet exercice sont dans le répertoire `exo3`.

Question 1

Sur quelle structure de données présentée en cours sont fondées les piles ?

Solution:
Les listes chaînées.
Explication(s):

- ✿ Voir l'API des ensembles pour montrer la définition du type `integer_stack_t` qui n'est qu'un `typedef`.
- ✿ En conséquence nous pouvons utiliser les itérateurs sur les listes pour faire l'affichage d'une pile.

Question 2

Combien de variables de type pile allez-vous utiliser ?

Solution:
Il est possible d'utiliser deux piles pour répondre à ce problème p_1 ett p_2.
Explication(s):

- ✿ L'idée est alors d'utiliser une pile p_1 pour chaque appel à la fonction `enqueue` ;
- ✿ Et on transfert les données d'une pile p_1 à l'autre p_2 quand on fait appel à la fonction `dequeue` (à condition que p_2 soit vide).
- ✿ Il faut donc une fonction auxiliaire qui fait passer les donner d'une pile à l'autre.

Question 3

Donnez le nouveau type mettant en œuvre une file avec des piles.

Solution:

```
typedef struct myqueue_ {
  integer_stack_t in;
  integer_stack_t out;
} myqueue_t;
```

Question 4

Donnez la définition de la fonction `enqueue`.

Solution:

```
int enqueue (myqueue_t *queue, int value) {
  return integer_stack_push (&(queue->in), value);
}
```

Question 5

Donnez la définition de la fonction `dequeue`.

Solution:

```
int shift_stacks (integer_stack_t *in, integer_stack_t *out) {
  if (integer_stack_size (out) == 0) {
    int elt;
    while (integer_stack_size(in) > 0) {
      integer_stack_pop(in, &elt);
      integer_stack_push(out, elt);
    }
  }
  return 0;
}

int dequeue (myqueue_t *queue, int *value) {
  shift_stacks (&(queue->in), &(queue->out));
  return integer_stack_pop (&(queue->out), value);
}
```

Explication(s):

- ✿ C'est simplement dans cette fonction que nous utilisons la fonction auxiliaire qui fait passer les données d'une pile à une autre, cf `shift_stacks`.

Question 6
Donnez la définition de la fonction `size`.

Solution:

```
int size (myqueue_t *queue) {
  return integer_stack_size(&(queue->in)) + integer_stack_size (&(queue->out));
}
```

Question 7
Discuter de la complexité des opérations `enqueue` et `dequeue` et dire pourquoi ce n'est pas une bonne solution.

Solution:
- `enqueue` a une complexité constante $\mathcal{O}(1)$
- `dequeue` a une complexité linéaire $\mathcal{O}(n)$ avec n le nombre d'éléments dans la file. Au pire cas les éléments de la file sont dans la première pile et le seconde pile est vide. Il faut donc transférer tous les éléments dans la seconde pile.

POUR S'ENTRAÎNER À LA MAISON

Exercice 1 – Propriété d'unicité dans les ensembles
Cet exercice a pour objectif de manipuler la structure de données des ensembles (sets).

Remarque : Les données de cet exercice sont dans le répertoire `exo4`.

Question 1
Sur quelle structure déjà présentée en cours sont fondées les ensembles (sets) ?

Solution:
Les listes chaînées.
Explication(s):
- ✿ Voir l'API des ensembles pour montrer la définition du type `integer_set_t` qui n'est qu'un `typedef`.
- ✿ On peut donc utiliser les itérateurs sur les listes pour afficher le contenu d'un ensemble.

Question 2
L'objectif de cet exercice est d'utiliser un ensemble (set) pour lister les différents caractères alphabétiques qui composent une phrase (on ne considérera donc pas les espaces et les symboles de ponctuation) et on normalisera les caractères pour ne conserver que les minuscules.

Décrivez un algorithme pour faire cela s'appuyant sur la structure de données d'ensembles (sets) et mettez le en œuvre.

Solution:

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

#include "set.h"
#include "list.h"

int main (){

  char* str = "Salut les amis, je vous aime de tout mon coeur.";
  int len = strlen(str);

  character_set_t letters;
  character_set_init (&letters);

  for (int i = 0; i < len; i++) {
    /* keep only alphabetic characters */
    if (isalpha(str[i])) {
      /* keep only lowercase version of characters */
      character_set_insert (&letters, tolower(str[i]));
    }
  }

  /* set is an alias for list so cast to define iterator */
  character_list_t letters_l = letters;
  character_list_elmt_t *elt = character_list_head(&letters_l);
  for (; elt != character_list_tail(&letters_l); elt = character_list_next (elt)) {
    printf ("%c, ", character_list_data(elt));
  }
  printf ("%c\n", character_list_data(character_list_tail(&letters_l)));

  character_set_destroy (&letters);

  return EXIT_SUCCESS;
}
```

Explication(s):

- ✿ Rappelez le fonctionnement d'un ensemble ;
- ✿ Algorithme simple, on parcours la chaîne de caractère, on insère dans l'ensemble que les caractères alphabétiques en minuscules ;
- ✿ on utilise les fonctions `isalpha` et `tolower` de la bibliothèque `ctype.h` pour ne conserver que les caractères alphabétiques et les transformer en minuscules.

Exercice 2 – Tri radix

Il existe de nombreux algorithmes de tri de données comme *bubble sort*, *merge sort*, *quick sort*, etc. La complexité de ces algorithmes oscille entre $\mathcal{O}(n^2)$ et $\mathcal{O}(n\log(n))$ où n est le nombre d'éléments à trier.

Dans cet exercice nous allons explorer une autre méthode de tri, nommée *tri radix* ou *tri par base*. Cet algorithme consiste à trier les nombres chiffres par chiffre. Le principe est le suivant :

— On trie tous les nombres par rapport aux chiffres des unités
— Puis on trie les nombres par rapport aux dizaines, puis par rapport aux centaines, etc.

Remarque : Les données de cet exercice sont dans le répertoire `exo2`.

Question 1

Écrivez une fonction `find_max` dont le prototype est :

```c
int find_max(integer_list_t *list);
```

Solution:

```
int find_max (integer_list_t *list) {
  int max = INT_MIN;
  integer_list_elmt_t* elt = integer_list_head(list);
  for (; elt != integer_list_tail(list); elt = integer_list_next(elt)) {
    int data = integer_list_data(elt);
    if (data > max) {
      max = data;
    }
  }
  return max;
}
```

Explication(s):

- On reprend la structure de l'itérateur sur les listes utilisé dans l'exercice concernant l'affichage d'une liste.

- Pour utiliser la constante `INT_MIN` il faut inclure le fichier `limits.h`.

Question 2

Mettez en œuvre l'algorithme du tri par base dans une fonction dont le prototype est :

```
void radix_sort (integer_list_t* initial, integer_list_t* sorted, int max)
```

Solution:

```
void radix_sort (integer_list_t* initial, integer_list_t* sorted, int max) {
  /* Copy initial into sorted  */
  integer_list_elmt_t* elt = integer_list_head(initial);
  for (; elt != NULL; elt = integer_list_next(elt)) {
    integer_list_ins_next (sorted, integer_list_tail(sorted),
                           integer_list_data(elt));
  }

  for (int exp = 1; exp <= max; exp *= 10) {
    integer_list_t temp;
    integer_list_init (&temp);
    for (int i = 0; i < 10; i++) {
      integer_list_elmt_t* elt = integer_list_head(sorted);
      for (; elt != NULL; elt = integer_list_next(elt)) {
        int data = integer_list_data (elt);
        if ( (data / exp) % 10 == i) {
          integer_list_ins_next (&temp, integer_list_tail(&temp), data);
        }
      }
    }

    /* Copy temp into sorted after clearing sorted */
    integer_list_destroy(sorted);
    integer_list_init(sorted);

    integer_list_elmt_t* elt = integer_list_head(&temp);
    for (; elt != NULL; elt = integer_list_next(elt)) {
      integer_list_ins_next (sorted, integer_list_tail(sorted),
                             integer_list_data(elt));
    }
    /* A new temp list is created at each iteration */
  }
}
```

Explication(s):

- La fonction prend en paramètre la liste à trier et la liste triée qui sera retournée.
- Il y a deux blocs de code dans cette fonction.
  1. une copie de la liste initiale dans la seconde liste
  2. une boucle qui itère sur toutes les puissance de 10 permettant d'accéder aux différentes positions des chiffres (unité, dizaine, etc.)
  3. une boucle qui itère sur tout les chiffres (de 0 à 9)
  4. une boucle interne qui classe les nombres en fonction du chiffre considéré. On utilise ici une liste intermédiaire qu'il faut copier régulièrement dans la liste résultat `sorted`.

Question 3

Discuter de la complexité pire cas de cet algorithme et comparez la par rapport aux autres algorithmes.

Solution:
La complexité de cet algorithme est $\mathcal{O}(n\omega)$ avec n le nombres d'éléments dans la liste (boucle interne) et ω le nombre maximal de digits (boucle externe).

APPROFONDISSEMENT

Exercice 1

L'objectif de cet exercice est de mettre en oeuvre une extension de la bibliothèque `libin103` pour gérer les tables de hachage. Pour corser le tout, nous allons considérer la version générique de cette structure de données.

Nous supposons travailler dans le répertoire `exo6` pour cet exercice.

Eléments sur les tables de hachage Les listes chaînées sont une structure de données interressantes pour stocker et manipuler des données dynamiquement sans connaissance *a priori* du nombre d'éléments à stocker. Cependant, la complexité linéaire de la recherche d'un élément ou au mieux une complexité logarithmique dans le cas d'une liste chainée triée, peut rendre son utilisaton peu aisée dans certains cas.

Les tableaux sont une structure de données qui permet un adressage direct aux éléments mais ne permettent d'indexer des éléments que par des entiers.

Les tables de hachages permettent de représenter des ensembles de couples (clef, valeur) avec un accès rapide. L'ensemble des clefs étant grand, l'idée est de plongée cet espace dans un espace plus petit. On utilise un tableau pour représenter des tables de hachage mais les indices du tableau sont calculés à partir de la clef initiale qui est "hachée", associée à un entier. Une problématique dans les tables de hachage est que plusieurs clefs peuvent être associées au même indice du tableau. On parle alors de collision, plusieurs solutions sont possibles pour gérer ces conflits. La solution que nous adopterons pour cet exercice est l'utilisation de liste chaînée pour accumuler au niveau d'un même indice du tableau l'ensemble des couples (clefs, valeurs) qui dont la clef est "hachée" de la même manière.

Pour plus d'explications, vous pouvez regarder le cours de M. Pessaux sur les tables de hachage

`https://perso.ensta-paris.fr/~chapoutot/teaching/in103/slides/cours-table-hachage.pdf`

Travail à faire L'API que nous proposons de développer pour les tables de hachage est donnée dans le fichier `exo6/generic_hashtable.h`. Il consiste en des définitions de types, une fonction d'initialisation et de destruction ainsi que des fonctions d'ajout et de suppression.

Question 1

Programmez les fonctions associées à cette API dans fichier nommé `generic_hashtable.c`

Solution:
Cf code source donné dans le repretoire exo6 de la correction. Le fichier est trop long pour être affiché ici.