Patterns Algorithmiques
TD * - Motifs récurrents

Ce document a pour objectif de vous rappeler les motifs récurrents nécessaires à quasiment n'importe quel algorithme. Vous devez connaître, reconnaître, et savoir utiliser l'ensemble de ces motifs (appelés « *patterns* » en anglais). Les patterns étant relativement courts, ils ne sont que la combinaison de quelques mots clés afin d'effectuer une micro-tâche extrêmement simple (**`if`**, **`while`**, ...).

Le document est divisé en deux parties : la première présente d'abord en langage humain chaque pattern ainsi que le code associé, puis, dans la deuxième partie, des problèmes avec leurs méthodes de résolution sont présentés.

*Il est **nécessaire** de maîtriser l'ensemble de ces patterns à l'issue de la première année de n'importe quelle formation en informatique.*

1 Pattern -> Code

1.1 Prototypes

1.1.1 Tableaux en paramètres

Un *tableau* doit toujours être accompagné de sa *taille*, sauf, si le langage de programmation cible a des dispositifs embarquant la taille avec le tableau (ce qui n'est pas le cas du C en l'absence de structure).

```c
void ma_fonction1(int tab[], int size);
void ma_fonction2(int *tab,  int size);
```

Une *chaîne de caractères* (ou *string* en anglais) est un tableau contenant des caractères se terminant par un « **`\0`** ». Selon les besoins : si le développeur doit *renvoyer* une chaîne de caractères, c'est à lui d'allouer suffisamment d'espace en mémoire **et** d'ajouter le « **`\0`** » final.

```c
void ma_fonction(char *str);
```

À l'inverse, un *tableau de caractères* n'est pas nécessairement terminé par un « **`\0`** », donc sa taille *doit* être indiquée en paramètre.

```c
void ma_fonction1(char tab[], int size);
void ma_fonction2(char *tab,  int size);
```

1.1.2 Types de retour

Concernant les types de retour : réfléchissez à l'énoncé des questions/du problème. Si vous devez renvoyer une *taille*, il s'agit d'une *quantité*, donc d'un *entier* ou d'un *flottant*. Pour déterminer parmi les deux, c'est à vous de lire le problème en détails pour comprendre s'il s'agit d'un cas discret (entiers) ou continu (flottants).

La taille d'une chaîne de caractères implique de compter le nombre de lettres : cas discret, donc on renvoie un entier. Le nombre de chiffres constituant un nombre, même chose : cas discret, donc on renvoie un entier. Par contre, la taille d'un individu, un premier cas implique la valeur continue sous forme de flottant ($1,72$ m), un second cas peut être un entier (entre 0 et 3 mètres) et un autre entier (entre 0 et 99 décimètres) formant une combinaison (1m72).

1.2 Patterns de tableaux

Un tableau n'étant qu'un vecteur à une dimension, les principaux besoins consistent à :

1. itérer jusqu'à la fin ou jusqu'à un critère
2. décaler chaque élément du tableau un cran à gauche/droite
3. faire de la reprise d'itération à partir d'un contexte existant

1.2.1 Itérer jusqu'à la fin/un critère

Objectif : avancer de case en case pour effectuer un traitement

Le tableau n'étant qu'un vecteur, et connaissant sa taille, il suffit juste d'effectuer une boucle pour atteindre la fin :

```c
void ma_fonction(char tab[], int size)
{
  int i;
  for (i = 0; i < size; i++)
  {
    printf("Valeur : %d\n", tab[i]); // traitement
  }
}
```

On peut égalementer itérer selon un critère d'arrêt (les 10 premières cases) :

```c
void ma_fonction(char tab[], int size)
{
  int i = 0;
  while (i < 10)
  {
    printf("Valeur : %d\n", tab[i]); // traitement
    i += 1;
  }
}
```

Attention ! Si le tableau ne contient pas le critère, vous **devez** vous arrêter avant la fin ! (donc ajouter un deuxième critère : celui d'arrêt). L'ordre des tests est important : on teste *d'abord* si l'on est encore dans le tableau, et si oui, on peut regarder si le critère est vérifié ou non :

```c
void ma_fonction(char tab[], int size)
{
  int i = 0;
  while ((i < size) && (i < 10))
  {
    printf("Valeur : %d\n", tab[i]); // traitement
    i += 1;
  }
}
```

1.2.2 Décaler chaque élément du tableau un cran à gauche/droite

Objectif : décaler chaque élément de la case n vers la case $n \pm 1$

Pour décaler chaque élément, il faut réfléchir au sens : si l'on veut tout décaler vers la droite, on démarrer de la fin du tableau vers le débutet ramener chaque élément (sinon la même valeur est écrite partout). De plus, il faut faire attention aux extrêmités du tableau : si on décale vers la droite et que l'on démarre par la dernière case, on doit s'arrêter *avant* la case 0 (car la case 1 va copier le contenu de la case 0, mais la case 0 ne peut pas copier la case « -1 » qui n'existe pas).

```
void decale_a_droite(char tab[], int size)
{
  int i;
  for (i = (size - 1); i > 0; i--)
  {
    tab[i] = tab[i - 1];
  }
}
```

```
void decale_a_gauche(char tab[], int size)
{
  int i;
  for (i = 0; i < size; i++)
  {
    tab[i] = tab[i + 1];
  }
}
```

1.2.3 Reprise d'itération à partir d'un contexte

Objectif : itérer, retourner une valeur, redémarrer à partir de la dernière itération réalisée

La reprise d'itération est plutôt rare, mais elle est relativement importante. Pour être réalisée, il faut utiliser un paramètre qui servira de contexte (ce paramètre peut être modifié par la fonction, ou, être directement la valeur retournée). Le cas d'arrêt reste d'avoir atteint la fin du tableau (ou même de l'avoir dépassé si un contexte incorrect est donné en paramètre).

```
void reprise(char tab[], int size, int last)
{
  if (last >= size)
    return (size);

  int i;
  for (i = last; i < size; i++)
  {
    if (tab[i] % 2 == 0)
      return (i);
  }
  return (size);
}
```

1.3 Patterns de code

Plusieurs problèmes récurrents arrivent dans quasiment tous les algorithmes :

1. Détecter ou filtrer des valeurs spéciales
2. Ordonner les tests (y compris en récursif)

1.3.1 Détecter/Filtrer des valeurs spéciales

Objectif : retirer ou transformer des valeurs pouvant entraîner des problèmes

Détecter des valeurs est l'opération élémentaire de *test d'une condition* (donc un **`if`**). Une fois la valeur détectée, plusieurs actions sont possibles : arrêter le programme en renvoyant une erreur ou en déclenchant une exception, transformer la valeur problématique en une valeur par défaut, ou, dans le cas des boucles, ignorer la valeur/passer à l'itération suivante (avec le mot clé **`continue`**).

```c
int fonction(int value)
{
  if (value == -42)
    printf("! ERREUR ! [value=%d]\n", value);
    return (-1);

  printf("Valeur : %d\n", value);
  return (0);
}
```

```c
void fonction(int value)
{
  if (value < 1)
    value = 0;
  if (value > 99)
    value = 100;

  printf("Valeur [0-100] : %d\n", value);
}
```

```c
void fonction(int tab[], int size)
{
  for (int i = 0; i < size; i++)
  {
    if (tab[i] < 0)
      continue ;

    printf("Valeur : %d\n", tab[i]);
  }
}
```

1.3.2 Ordonner les tests (récursif ou non)

Objectif : ne pas faire planter la récursion ni rater de test conditionnel

L'ordre des tests est relativement important : certains tests peuvent ne jamais être effectués s'ils sont mal imbriqués/mal placés dans le code. Par exemple : on ne doit pas tester une valeur spécifique positive dans un test d'une valeur négative (jamais une valeur positive ne pourra arriver là). Dans le cas des appels récursifs, il est nécessaire de distinguer les cas d'erreurs/pouvant entraîner des plantages *avant* de tester les valeurs recherchées, et seulement après ces deux tests on peut tester des cas plus génériques.

```c
int fonction(int value)
{
  if (value > 0)
  {  // Positif
    if (value == 42)
      printf("Quarante Deux !!!\n");
    return (1);
  }
  else
  {  // Negatif
    if (value == -42)
      printf("Moins Quarante Deux !!!\n");
    return (-1);
  }
}
```

```c
int fonction(int tab[], int size, int value)
{
  // Cas d'erreur
  if ((size < 1) || (value < 1))
    return (-1);

  // Cas d'arret
  if (value < 10)
    return (1);

  // Cas general
  return (1 + fonction(tab, size, (value / 10)));
}
```

1.4 Patterns dans des structures de données

Les structures de données possèdent également des cas récurrents :

1. Listes : atteindre la fin d'une liste
2. Listes : rechercher une valeur/s'arrêter au milieu

1.4.1 Listes : atteindre la fin d'une liste

Objectif : s'arrêter sur le dernier élément d'une liste

Accéder au dernier élément d'une liste est une opération relativement simple mais nécessaire dans certains cas. Il faut surtout faire attention à vérifier que la liste n'est pas vide. Selon le format de la liste, ainsi que le langage de programmation employé, plusieurs algorithmes sont possibles.

Liste à tableau Dans le format tableau, il suffit simplement d'utiliser l'index de fin de liste (s'il existe), ou d'aller à la dernière case utilisée.

```
struct liste_t
{
  int *array;
  int size;
  int last_index;
};

int get_last_elt(struct liste_t *list)
{
  int value;

  if (list->size == 0)
    return (-1);

  value = list->array[list->last_index];
  return (value);
}
```

Liste chaînée Dans le format liste chaînée (à pointeurs), il est nécessaire de dérouler la liste jusqu'au bout. Il faut néanmoins s'arrêter juste avant l'éventuel pointeur de fin. Attention au cas où la liste est déjà vide.

```
struct liste_p
{
  int value;
  struct liste_p *next;
};

struct liste_p *get_last_elt(struct liste_p *list)
{
  struct liste_p *tmp;

  if (list == NULL)
    return (NULL);

  tmp = list;
  while (tmp->next != NULL)
    tmp = tmp->next;

  return (tmp);
}
```

Liste circulaire Si la liste est circulaire, il faut également s'intéresser au format sous-jacent : en tableau, il faut simplement regarder le pointeur de fin, en pointeur, il suffit de revenir un cran en arrière. Dans tous les cas, il faut tester auparavant que la liste n'est pas vide.

1.4.2 Listes : rechercher une valeur/s'arrêter au milieu

Objectif : s'arrêter sur un élément précis (désigné soit par sa valeur, soit par son index)

Accéder à un élément précis d'une liste est une opération simple. Il faut non seulement s'assurer que la liste n'est pas vide, mais également rechercher l'élément. Selon le format de la liste, ainsi que le langage de programmation employé, plusieurs algorithmes sont possibles.

Liste à tableau Dans le format tableau, on réutilise directement le pattern d'itération jusqu'à un critère précis : l'égalité entre l'élément recherché et celui contenu OU le numéro d'index. On n'oubliera pas de tester d'abord si l'on a atteint la fin de la liste ou pas.

```
struct liste_t
{
  int *array;
  int size;
  int last_index;
};

int get_elt_by_value(struct liste_t *list, int value)
{
  if (list->size == 0)
    return (-1);

  int i = 0;
  while ((i < list->last_index) && (list->tab[i] != value))
    i++;

  if (list->tab[i] == value)
    return (i);
  else
    return (-1);
}
```

```
int get_elt_by_index(struct liste_t *list, int index)
{
  if ((list->size == 0) || (index > list->last_used))
    return (-1);

  int value;
  value = list->tab[index];
  return (value);
}
```

Liste chaînée Dans le format liste chaînée (à pointeurs), il est nécessaire de dérouler la liste jusqu'à l'élément souhaité. La longueur de la liste n'étant pas connue à l'avance (sauf si la structure l'embarque), il est nécessaire de s'assurer que l'on n'a pas dépassé la fin. Attention au cas où la liste est déjà vide.

```
struct liste_p
{
  int value;
  struct liste_p *next;
};

struct liste_p *get_elt_by_value(struct liste_p *list, int value)
{
  struct liste_p *tmp;

  if (list == NULL)
    return (-1);

  int i = 0;
  tmp = list;
  while ((tmp != NULL) && (tmp->value != value))
  {
    i += 1;
    tmp = tmp->next;
  }

  if (temp == NULL)
    return (-1):
  return (i);
}
```

```
struct liste_p *get_elt_by_index(struct liste_p *list, int index)
{
  struct liste_p *tmp;

  if (list == NULL)
    return (-1);

  int i = 0;
  tmp = list;
  while ((tmp != NULL) && (i != index))
  {
    i += 1;
    tmp = tmp->next;
  }

  if (temp == NULL)
    return (-1):
  return (tmp->value);
}
```

2 Problèmes et patterns

Pour chaque problème listé, vous devez indiquer le ou les patterns nécessaires pour les traiter. Indiquer également le prototype *complet* de chaque fonction permettant de traiter le problème (c'est-à-dire le type de retour, ainsi que les paramètres et leurs types) lorsqu'elle est indiquée. Lorsque deux implémentations ou plus peuvent exister (par exemple en tableau et en maillons/pointeurs), indiquer les patterns pour chaque version.

1. nombre de chiffres dans un nombre
2. nombre de chiffres dans un nombre dans une base quelconque
3. **`is_mirroir`** : test si un nombre est un miroir
4. **`miroir`** : construire le miroir d'un nombre
5. **`strlen`** : atteindre la fin d'une chaîne de caractères
6. **`is_palindrome`** : test si une chaîne est un palindrome
7. **`itoa`** : transformation d'un entier en chaîne de caractères
8. **`atoi`** : transformation d'une chaîne de caractères en un entier
9. **`min/max_tab`** : valeur minimale/maximale d'un tableau
10. **`sum_tab`** : somme des éléments d'un tableau
11. **`croissant_tab`** : vérification qu'un tableau est trié dans l'ordre croissant
12. **`compare_tabs`** : comparer deux tableaux
13. **`inverse_tab`** : inverser la position de tous les éléments d'un tableau
14. **`substr`** : recherche d'une sous-chaîne dans une chaîne de caractères
15. **`bubble_sort`** : algorithme de tri à bulles
16. **`remove_elt_in_list`** : suppression d'un élément dans une liste
17. **`revert_list`** : inversion de la position de tous les éléments d'une liste
18. **`clear_list`** : suppression de tous les éléments d'une liste

Ce document et ses illustrations ont été réalisés par Fabrice BOISSIER en mars 2025. Certains exercices sont inspirés des supports de cours de Nathalie "Junior" BOUQUET, et Christophe "Krisboul" BOULLAY.