Algorithmes et structures de données : **NF16**

Automne 2015

Table des matières

1 Rappels de C **2**
1.1 Gestion de la mémoire . . . 2
1.1.1 Pointeurs . . . 2
1.1.2 La mémoire en C . . . 2
1.1.3 Gestion dynamique de la mémoire / malloc(), free() . . . 3
1.1.4 Passage de paramètres, modification d'objets à travers les fonctions . . . 4
1.2 typedef, structures . . . 4

2 Analyse d'algorithmes **6**
2.1 Caractéristiques d'un algorithme . . . 6
2.1.1 Spécification d'un algorithme . . . 6
2.1.2 Problème d'un algorithme . . . 6
2.2 Complexité d'un algorithme . . . 6
2.2.1 Propriétés utiles au calcul de la complexité . . . 6
2.3 Preuve d'un algorithme . . . 7

3 Structures de données linéaires **8**
3.1 Tableaux . . . 8
3.1.1 non triés . . . 8
3.1.2 triés . . . 8
3.2 Listes chaînées - simple/double . . . 8
3.3 Files / Structures First In First Out . . . 8

4 Structures de données arborescentes **9**
4.1 Arbres binaires . . . 9

Chapitre 1

Rappels de C

1.1 Gestion de la mémoire

1.1.1 Pointeurs

Il est utile de d'abord rappeler ce que sont les pointeurs en C :

- Un pointeur se déclare sous la forme <type>* <nomDuPointeur> ; il doit obligatoirement pointer sur une variable.
- Un pointeur comporte l'adresse de la variable sur laquelle il pointe.
- Afin d'accéder au contenu de la variable sur laquelle le pointeur pointe, il faut utiliser *<*pointeur*>.
- Utiliser <pointeur> permettra d'accéder à l'adresse de la variable pointée.

En gros, un pointeur est une adresse, plus précisément celle de la variable pointée.

1.1.2 La mémoire en C

Il existe, en gros, 3 zones importantes de mémoire pour chaque programme C :

- *data segment*, qui contient toutes les données statiques et globales du programme ;
- *heap* ou *tas*, qui est destinée aux données globales allouées dynamiquement ;
- *stack* ou la *pile*, qui contient un ensemble de boites - *frames* - qui contiennent, pour chaque boite, toutes les données locales/temporaires à chaque fonction, ainsi que les paramètres de fonction ;

Ainsi, l'emplacement d'une donnée/variable dépend de la manière dont l'espace mémoire lui a été réservée, et cela a des conséquences sur la portée de la variable, ainsi que sur sa "durée de vie".
Dans le code suivant, on a des variables qui se trouvent à différents endroits de la mémoire en fonction de la manière dont ils ont étés déclarés/initialisés.

```
/* Ces deux variables se trouvent dans le data segment */
static int variableData = 2;
int variableBSS = 3;
void foo()
{
        /* Se trouve dans la pile, dans une boite "foo"
         * Est détruit à la fin de l'exécution de foo() */
        int i = 1;
}

int main()
{
```

```
        /* Se trouve dans le tas, est accessible pour n'importe quelle
           fonction */
        int* pointeurEntier;
        pointeurEntier = (int*) malloc(sizeof(int));
        *pointeurEntier = 4;
        /* Se trouve dans la pile, dans une boite "main" */
        int entier = 5;
        /* Une boite "foo" est créée au dessus de la boite "main" */
        foo();
        return 0;
}
```

On le voit dans cet exemple que toutes les données ne se trouvent donc pas au même endroit.
Lorsque on déclare une variable, par exemple avec *int i=3*, cette variable sera automatiquement placée dans la pile, plus exactement dans la "boite" de la fonction. Ainsi, elle sera automatiquement supprimée à la fin de l'exécution de la fonction, ce qui est fâcheux si jamais on cherche par exemple à construire un objet qu'on veut réutiliser au delà de la fonction ...
Dans le code suivant, par exemple :

```
Objet* newObjet()
{
        Objet newObj;
        newObj.attribut = 1;
        [...]
        return &newObj;
}
```

On cherche ici à retourner l'adresse du nouvel Objet *newObj*. Le problème, c'est que l'espace mémoire alloué au nouvel objet sera **automatiquement détruit** à la fin de la fonction. On renvoie donc une adresse/-pointeur qui est faux, et qui pointera sur des données incorrectes. Pour remédier à cela, il faut recourir à la gestion *dynamique* de la mémoire.

1.1.3 Gestion dynamique de la mémoire / malloc(), free()

malloc() retourne un pointeur vers un espace mémoire alloué dans le tas.
Cette fonction prend en paramètre une taille en éléments mémoire, qui correspond à la taille de l'espace mémoire à allouer. Par exemple, en reprenant l'objet *Objet*, on pourrait allouer un espace mémoire avec malloc de la manière suivante :

```
Objet* newObjet()
{
    Objet* pointeurObj = (Objet*) malloc(sizeof(Objet));
    pointeurObj->attribut = 1;
    return pointeurObj;
}
```

On récupère alors l'adresse du nouvel objet.

Il ne faut pas oublier de déallouer / libérer la mémoire qui a dynamiquement été louée.
En effet, si on libère pas cet espace, il ne le sera jamais, ce qui peut entraîner des fuites mémoires.
Pour libérer la mémoire, il faut alors utiliser la fonction *free()*, qui prend en paramètre un pointeur vers l'espace mémoire à libérer.
Toujours en prenant l'exemple précédent, on libère l'espace utilisé de la manière suivante :

```
// On récupère un nouvel objet alloué dynamiquement
Objet* obj = newObjet();
// On libère l'espace
free(obj);
```

1.1.4 Passage de paramètres, modification d'objets à travers les fonctions

Comme écrit précédement, les paramètres passés par valeur sont copiés dans la pile.
Par exemple, dans le code suivant :

```
void func1(int i)
{
        i = 4;
}

void func2()
{
        int i = 2;
        func1(i);
        printf("%i",i);
}
```

On pourrait croire que la valeur de *i* dans *func2* serait modifiée. Il en est rien, puisque *i* est en fait copié dans la "boite" de la fonction "func1". C'est donc une **copie locale** de la variable *i* qui est modifiée dans *func1*.

En réalité, pour modifier une valeur à travers les fonctions, il faut passer en paramètre u pointeur, ou une adresse mémoire vers le contenu à modifier.
En reprenant l'exemple d'en haut, il faudrait alors écrire :

```
void func1(int* i)
{
        *i = 4;
}

void func2()
{
        int i = 2;
        func1(&i);
        printf("%i", i);
}
```

Dans ce cas là, on a bien modifié *i*, et on affiche donc *4* dans *func2*.
C'est un *passage par pointeur*, à constrater avec le *passage par valeur*.

1.2 typedef, structures

Les *structures* en C ressemblent un peu aux *objets* que l'on rencontre souvent dans des langages orientés objets, tel que le Java.
Ils sont composés d'attributs, et se déclarent de la manière suivante :

```
struct nomStructure {
        [attributs]
};
```

On peut ensuite les appeller avec *struct nonStructure*.
Il est aussi possible de faire des *typedef*, qui permet de se débarasser de ce *struct* à chaque appel :

```
typedef struct {
        [attributs]
} nomStructure;
```

Chapitre 2

Analyse d'algorithmes

2.1 Caractéristiques d'un algorithme

2.1.1 Spécification d'un algorithme

La spécification d'un algorithme correspond à décrire ce que l'algorithme fait sans expliquer comment il le fait.
Cependant, une spécification se doit d'être précise. Ainsi, il faut expliciter quelles sont les données, les valeurs valables, comment se termine l'algorithme et ce qu'il retourne.

2.1.2 Problème d'un algorithme

Un algorithme doit prendre plusieurs facteurs en compte lors de son exécution :
- Une instance du problème, qui fournit toutes les données nécessaires à l'exécution de l'algorithme ;
- Une taille du problème, qui caractérise le nombre de données que l'algorithme manipule ;

Enfin, un algorithme est aussi caractérisé par sa vitesse d'exécution, et la place qu'il occupe lors de l'exécution : ce sont les complexités *temporelles* et *spatiales*.

2.2 Complexité d'un algorithme

La complexité d'un algorithme se mesure avec le comportement asymptotique [1] de ce dernier.
On mesure la complexité dans deux cas différents :
- Dans le **pire des cas**, noté θ ;
- Dans le **meilleur des cas**, noté Ω ;

θ permet de fixer une borne limite haute, tandis qu'Ω fixe une borne limite basse.
Lorsque $\theta = \Omega$, on connait alors la complexité exacte, notée Φ.

2.2.1 Propriétés utiles au calcul de la complexité

Inclusions
- $f(n) \in \theta(g(n)) \Leftrightarrow \exists c, n_0$ tels que $\forall n \geq n_0, 0 \leq f(n) \leq g(n)$
- $Si \lim_{n\to\infty} f(n)/g(n) = \infty \Leftrightarrow f(n) \notin \theta(g(n))$ et $g(n) \in \theta(f(n))$
- $Si \lim_{n\to\infty} f(n)/g(n) = \lambda, \lambda \in \mathbb{R} \Leftrightarrow f(n) \in \theta(g(n))$ et $g(n) \in \theta(f(n))$
- $Si \lim_{n\to\infty} f(n)/g(n) = 0, \Leftrightarrow f(n) \in \theta(g(n))$ et $g(n) \notin \theta(f(n))$

1. au voisinage de l'infini

Règle du maximum

— Si f, g tendent vers $\mathbb{R}+$, on a alors $\theta(f(n) + g(n)) = \theta(max(f(n), g(n)))$

2.2.2 Coût des opérations dans un algorithme

Selon la composition de l'algorithme, ce dernier peut avoir une complexité plus ou moins grande :

— Opération élémentaire/Affectation : $\theta(1)$

— Bloc d'instruction : somme

— Si/Sinon : $\theta(max(f_si(n), f_sinon(n)))$

— Pour/Tant que/Repeter : $borne_suprieure_corpsBoucle * borne_suprieure_nombreItrations$

2.3 Preuve d'un algorithme

Prouver à algorithme consiste à montrer deux choses :

— Qu'il s'arrête ;

— Qu'il donne le bon résultat ;

Chapitre 3

Structures de données linéaires

3.1 Tableaux

3.1.1 non triés

3.1.2 triés

3.2 Listes chaînées - simple/double

3.3 Files / Structures First In First Out

Chapitre 4

Structures de données arborescentes

4.1 Arbres binaires