Correction des exercices de la session 3

1. (a) Si nous regardons la fonction sans s'occuper des appels récursifs, nous remarquons qu'elle a une complexité en temps et en espace constante. Ceci sera multiplié par le nombre de contextes on devra mettre sur la pile d'exécution. Le nombre de contextes simultanés dans la pile d'exécution est $\log_2 n + 2$. La complexité en espace est donc de $(\log_2 n + 2) \times 1$ ce qu'on approximise par $\log n$. Nous avons donc une complexité théorique en espace de $\mathcal{O}(\log n)$. Pour la complexité en temps, on va regarder le nombre d'appels total. Nous appelons $f(n/2)$ $\log_2 n$ fois et chaque fois $f(n\%2)$ 1 fois car ça correspond toujours à notre cas de base. Nous avons donc de nouveau une complexité en temps théorique de $\mathcal{O}(\log n)$.

   (b) Pour écrire ce genre de fonction, on récupère la cas de base dont l'inverse va devenir notre gardien de boucle et le changement des arguments dans les appels théoriques deviendront nos incréments dans la boucle :

```
unsigned f(unsigned n) {
    unsigned r;
    for (r = 0; n > 1; n/=2)
        r += n%2;
    r += n; \\Correspond a ce qu'on fait pour le cas de base.
    return r;
}
```

2. (a) Cette fonction compte le nombre de chiffres 9 il y a dans le nombre v donné. La difficulté de cette fonction vient de l'opérateur ternaire `? :`. Cette opérateur exprime que si l'expression à gauche du `?` est vraie, alors l'instruction à gauche de `:` s'effectue, sinon c'est l'instruction à droite de `:`. Ici, on a donc $n = 1$ si $v\%10 == 9$ et $n = 1$ sinon.

   (b) Sans appels récursifs, cette fonction a une complexité en temps et espace constante. La complexité théorique variera donc en fonction du nombre de contextes sur la pile d'exécution. Nous allons appeler cette fonction $\log_{10} n$ fois et il y aura donc $\log_{10} n + 1$ contextes sur la pile au total et simultanément. La complexité théorique en temps et en espace est donc de $\mathcal{O}(\log n)$

   (c) Voici une fonction :

```
unsigned f(unsigned v) {
    unsigned n;
    for (n = 0; !v; v/=10)
        if (v%10 == 9)
            n++;
    return n;
}
```

3. (a) Nous allons présenter deux versions. Une itérative et une récursive. L'itérative est une version améliorée où nous n'allons garder en mémoire chaque fois que 2 lignes vu qu'on a juste besoin de la précédente pour calculer l'acutelle. On pourrait le faire en créant et gardant en mémoire l'entièreté du triangle sous la forme d'une matrice. Ceci demanderait cependant plus de mémoire et demanderait d'allouer dynamiquement un tableau en deux dimensions ce qui est plus compliqué. Voici néanmoins un bout de code montrant comment allouer une matrice et comment la libérer :

```
unsigned ** t;
t = malloc(sizeof(unsigned *)*i);
for (unsigned j = 0; j < i; j++)
    t[j] = malloc(sizeof(unsigned)*i);
for (unsigned j = 0; j < i; j++)
```

```
    free(t[j]);
free(t);
```

Commençons par la version itérative :

```
unsigned * pascal(unsigned i) {
    unsigned *l1, *l2;
    l1 = malloc((i+1)*sizeof(unsigned));//Va contenir la ligne precedente
    if (!l1)
        return NULL;
    l2 = malloc((i+1)*sizeof(unsigned));//Va contenir la ligne actuelle
    if (!l2) {
        free(l1);
        return NULL;
    }

    for (unsigned j = 0; j < i + 1; j++) {
        //On est chaque fois a la ligne j
        l2[0] = 1; //(Correspond a t[j][0])
        l2[j] = 1; //(Correspond a t[j][j])
        for (unsigned k = 1; k < j; k++) {
            //On s'occupe des colonnes de la ligne j
            l2[k] = l1[k-1] + l1[k];
        }
        for (unsigned k = 0; k < j + 1; k++){
            //On recopie les elements de l2 dans l1
            l1[k] = l2[k];
        }
    }
    free(l1);
    return l2;
}
```

Version récursive :

```
unsigned * pascal (unsigned i) {
    unsigned *l;
    l = malloc(sizeof(unsigned)*(i+1))
    if (!l)
        return NULL;
    l[0] = 1;
    l[i] = 1;

    if (i == 0)
        return l;

    unsigned *p = pascal(i-1);\\Recupere la ligne precedente
    if(!p) {
        free(l);
        return NULL;
    }
    for (unsigned j = 1; j < i; j++) {
        l[j] = p[j-1] + p[j];
    }
```

```
    free(p);
    return l;
}
```

4. (a) Cette fonction parcourt une chaine de caractères s et compte le nombre de fois que le caractère c apparait.

   (b) Aucune variable autre r n'est créée et toutes les isntructions se font en temps constant. La complexité va donc varier en fonciton du nombre d'appels récursifs. Nous allons parcourir la totalité du tableau (on s'arrête quand on trouve un caractère de terminaison). La complexité theorique en temps et en espace est donc de $\mathcal{O}(n)$ où n correspond à la taille de la chaine de caractère présent dans s, ce qu'on peut également noter $\mathcal{O}(|s|)$.

   (c) Voici la fonction :

```
unsigned f(char *s, char c) {
    unsigned r;
    for (r = 0; *s; s++)
        if (*s == c)
            r++;
    return r;
}
```