TD 1

Récursivité (rappels et un peu plus loin)

Exercice 1 (Récursivité)**.**

1. Que calcule la fonction suivante (donnée en pseudo-code et en C) ?

Fonction Toto(n)

si $n = 0$ **alors**
  retourner 1;
sinon
  retourner $n \times$ Toto$(n-1)$;

```
/* Fonction toto en C */

unsigned int toto(unsigned int n){
  if (n == 0) return 1;
  return n * toto(n - 1);
}
```

2. La suite de Fibonacci est définie récursivement par la relation $u_n = u_{n-1} + u_{n-2}$. Cette définition doit bien entendu être complété par une condition d'arrêt, par exemple : $u_1 = u_2 = 1$. Écrire une fonction qui calcule et renvoie le n-ième terme de la suite de Fibonacci ($n \in \mathbb{N}^*$ donné en argument de la fonction).
3. Écrire une fonction récursive qui calcule le pgcd de deux nombres entiers positifs.
4. Que calcule la fonction suivante ?

Fonction Tata(n)

si $n \leq 1$ **alors**
  retourner 0;
sinon
  retourner $1 +$ Tata $\left(\left\lfloor \frac{n}{2} \right\rfloor\right)$;

```
/* Fonction tata en C */

unsigned int tata(unsigned int n){
    if (n <= 1) return 0;
    return 1 + tata(n / 2);
}
```

5. Il n'est parfois pas suffisant d'avoir un bon cas de base, voici un exemple. En C, que vaut `Morris(1, 0)` ?

```
int Morris(int a, int b) {
  if (a == 0) return 1;
  return Morris(a - 1, Morris(a, b));
}
```

Exercice 2 (La fonction 91 de McCarthy)**.**
Les fonctions récursives mêmes simples donnent parfois des résultats difficiles à prévoir. Pour s'en convaincre voici un exemple. Pour $n > 100$ la fonction 91 de McCarthy vaut $n - 10$. Mais pour $n \leq 100$? Tester sur un exemple... pas trop mal choisi, puis prouver le résultat en toute généralité.

Fonction Tata(n)

si $x > 100$ **alors**
  retourner $x - 10$;
sinon
  retourner
  McCarthy(McCarthy$(x + 11)$);

```
int McCarthy(int x)
{
  if (x > 100) return(x - 10);

  return McCarthy(McCarthy(x + 11));
}
```

Exercice 3 (Tours de Hanoï)**.**
On se donne trois piquets, p_1, p_2, p_3 et n disques percés de rayons différents enfilés sur les piquets. On s'autorise une seule opération : Déplacer-disque(p_i, p_j) qui déplace le disque du dessus du piquet p_i vers le dessus du piquet p_j. On s'interdit de poser un disque d sur un disque d' si d est plus grand que d'. On suppose que les disques sont tous rangés sur le premier piquet, p_1, par ordre de grandeur avec le plus grand en dessous. On doit déplacer ces n disques vers le troisième piquet p_3. On cherche un algorithme (en pseudo-code ou en C) pour résoudre le problème pour n quelconque.

L'algorithme consistera en une fonction Déplacer-tour qui prend en entrée l'entier n et trois piquets et procède au déplacement des n disques du dessus du premier piquet vers le troisième piquet à l'aide de Déplacer-disque en utilisant si besoin le piquet intermédiaire. En C on utilisera les prototypes suivants sans détailler le type des piquets, `piquet_t` ni le type des disques.

```
void deplacertour(unsigned int n, piquet_t p1, piquet_t p2, piquet_t p3);
void deplacerdisque(piquet_t p, piquet_t q); /* p --disque--> q */
```

1. Indiquer une succession de déplacements de disques qui aboutisse au résultat pour $n = 2$.
2. En supposant que l'on sache déplacer une tour de $n - 1$ disques du dessus d'un piquet p vers un autre piquet p', comment déplacer n disques ?
3. Écrire l'algorithme en pseudo-code ou en donnant le code de la fonction `deplacertour`.
4. Combien de déplacements de disques fait-on exactement (trouver une forme close en fonction de n) ?
5. Est-ce optimal (le démontrer) ?

Exercice 4 (Le robot cupide)**.**
Toto le robot se trouve à l'entrée Nord-Ouest d'un damier rectangulaire de $N \times M$ cases. Il doit sortir par la sortie Sud-Est en descendant vers le Sud et en allant vers l'Est. Il a le choix à chaque pas (un pas = une case) entre : descendre verticalement ; aller en diagonale ; ou se déplacer horizontalement vers l'Est. Il y a un sac d'or sur chaque case, dont la valeur est lisible depuis la position initiale de *Toto*. Le but de *Toto* est de ramasser le plus d'or possible durant son trajet.

On veut écrire en pseudo-code ou en C, un algorithme Robot-cupide(x, y) qui, étant donnés le damier et les coordonnées x et y d'une case, rend la quantité maximum d'or (gain) que peut ramasser le robot en se déplaçant du coin Nord-Ouest jusqu'à cette case. En C, on pourra considérer que le damier est un tableau bidimensionnel déclaré globalement et dont les dimensions sont connues.

A	B
C	D

1. Considérons quatre cases du damier comme ci-dessus et supposons que l'on connaisse le gain maximum du robot pour les cases A, B et C, quel sera le gain maximum pour la case D ?
2. Écrire l'algorithme.
3. Si le robot se déplace d'un coin à l'autre d'un damier carré 4×4 combien de fois l'algorithme calcule t-il le gain maximum sur la deuxième case de la diagonale ? Plus généralement, lors du calcul du gain maximum sur la case x, y combien y a-t'il d'appels au calcul du gain maximum d'une case i, j ($i \leq x$, $j \leq y$).

Corrigé

Correction de l'exercice 1.

1. Factorielle de son argument.
2.
```
int Fibonacci(n)
{
    if (n < 3) /* cas de base */
    {
        return 1;
    }
    return Fibonacci(n - 1) + Fibonacci(n - 2); /* /!\ double appel récursif */
}
```
   Vous pouvez en profiter pour dire rapidement que ce genre de doubles appels récursifs prennent du temps en montrant par exemple que pour calculer `Fibonacci(n)` il faut revenir `Fibonacci(n)` fois au cas de base (puisque le résultat est calculé comme une somme $1 + 1 + \ldots + 1$ où les 1 proviennent du cas de base, n'hésitez pas à dessiner un arbre). Et la suite croît très vite, pour $n = 31$ on est déjà au delà du million.
3. On utilise $\mathrm{pgcd}(a, b) = a$ si $b = 0$ et $\mathrm{pgcd}(a, b) = \mathrm{pgcd}(b, a \mod b)$.
```
unsigned int pgcd(unsigned int a, unsigned int b){
  if (a < b) return pgcd(b, a);
  if (b == 0) return a;
  return pgcd(b, a % b);
}
```
4. Pour $n > 0$, la fonction Tata(n) calcule $\lfloor \log n \rfloor$ en base 2.
5. L'exécution de cette fonction (appel) provoque une `Segmentation fault` En effet `Morris(1, 0)` lance le calcul de l'expression `Morris(0, Morris(1, 0))` qui induit le calcul préalable de la valeur de `Morris(1, 0)` et ainsi indéfiniment, jusqu'à ce que le programme ne puisse plus lancer de nouveaux sous calculs faute de mémoire disponible à cet effet. (Sur mon système ça s'arrête au bout de 255×1024 appels, ça doit être quelque chose comme 8 Mo alloués à la pile, avec 8 mots de 32 bits alloués à chaque appel mais il ne faut pas s'étendre sur la structure de la mémoire d'un processus, les étudiants de MIEF et de maths n'ont pas vu ce genre de choses).

Correction de l'exercice 2.

Note : c'est le John McCarthy du Lisp, pas celui de la *terreur rouge*.

Prenons 91 comme exemple (ce n'est pas le meilleur choix mais un des plus naturels au regard de l'énoncé) et notons f la fonction. Comme $91 < 100$ le résultat de $f(91)$ sera $f(f(91 + 11)) = f(f(102))$. Mais $f(102) = 92$ donc $f(91) = f(f(102)) = f(92)$. Maintenant $f(92) = f(f(92 + 11)) = f(f(103)) = f(93)$ et par le même raisonnement $f(93) = f(94) = \ldots = f(99) = f(f(110)) = f(100) = f(f(111)) = f(101) = 91$. Donc $f(91) = \ldots = f(100) = 91$. Finalement, pour 90 on a $f(90) = f(f(101)) = f(91) = 91$. Voyons ce que ça donne pour 89. On a $f(89) = f(f(100)) = f(91) = 91$.

On conjecture que $f(n)$ vaut 91 pour tout $n \leq 100$ (y compris les négatifs).

Commençons par prouver plus proprement que $f(n) = 91$ pour les onze entiers n compris entre 90 et 100.

Démonstration. On a $f(101) = 91$. On montre que $f(n) = f(n + 1)$ pour $90 \leq n \leq 100$. Si $90 \leq n \leq 100$ alors

$$\begin{aligned} f(n) &= f(f(n+11)) \text{ et } n+11 \geq 101 \\ &= f(n+11-10) \\ &= f(n+1) \\ &\dots \\ &= f(101) \end{aligned}$$

On montre alors par récurrence sur k que lorsque $90 - k \times 11 \leq n \leq 100 - k \times 11$, $f(n) = 91$. Ceci démontrera que $f(n) = 91$ pour $(n \leq 100)$.

Le cas de base $k = 0$ est démontré. Supposons l'assertion vraie pour k on montre qu'elle l'est encore pour $k+1$. Soit n un entier dans l'intervalle $[90 - (k+1) \times 11, 100 - (k+1) \times 11]$. Alors $n \leq 100$ donc $f(n) = f(f(n+11))$. Mais $f(n+11)$ est dans l'intervalle $[90 - k \times 11, 100 - k \times 11]$ donc par hypothèse de récurrence $f(n+11) = 91$ et ainsi $f(n) = f(91)$ qui est égal à 91 (cas de base). Finalement le résultat est prouvé pour tous les intervalles $[90 - k \times 11, 100 - k \times 11]$ et leur union ($[90, 100] \cup [79, 89] \cup [68, 78] \cup \dots$) correspond à l'ensemble des entiers inférieurs à 100. □

Correction de l'exercice 3.

1. Facile :

```
deplacerdisque(p1, p2);
deplacerdisque(p1, p3);
deplacerdisque(p2, p3);
```

2. En trois coups de cuillères à pot : on déplace la tour des $n-1$ disques du dessus du premier piquet vers le deuxième piquet (en utilisant le troisième comme piquet de travail), puis on déplace le dernier disque du premier au troisième piquet et enfin on déplace à nouveau la tour de $n-1$ disques, cette fois ci du deuxième piquet vers le troisième piquet, en utilisant le premier piquet comme piquet de travail.

3. Ce qui précède nous donne immédiatement la structure d'une solution récursive :

```
void deplacertour(unsigned int n, piquet_t p1, piquet_t p2, piquet_t p3){
  if (n > 0){
    /* tour(n) = un gros disque D et une tour(n - 1)          */
    deplacertour(n - 1, p1, p3, p2); /* tour(n - 1): p1 -> p2 */
    deplacerdisque(p1, p3);          /* D: 1 -> 3             */
    deplacertour(n - 1, p2, p1, p3); /* tour(n - 1): p2 -> p3 */
  }
}
```

4. On fait $u_n = 2u_{n-1} + 1$ appels avec $u_0 = 0$. On pose $v_n = u_n + 1$ Ce qui donne $v_n = 2v_{n-1}$ avec $v_0 = 1$. On obtient $v_n = 2^n$ d'où la forme close $u_n = 2^n - 1$.

5. Par récurrence. Ça l'est pour $n = 0$. On suppose que ça l'est pour une tour de $n-1$ éléments. Supposons qu'on ait une série de déplacements quelconque qui marche pour une tour de n éléments. Il faut qu'à un moment m on puisse déplacer le disque du dessous. On doit donc avoir un piquet vide p pour y poser ce disque et rien d'autre d'empilé sur le premier piquet (où se trouve le "gros disque"). Le cas où p est le troisième piquet nous ramène immédiatement à notre algorithme. Dans ce cas, entre le début des déplacements et le moment m ainsi qu'entre juste après le moment m et la fin des déplacements on déplace deux fois en entier une tour de taille $n-1$. Notre hypothèse de récurrence établie que pour un seul de ces déplacements complet d'une tour on effectue au moins $2^{n-1} - 1$ déplacements de disques. En ajoutant le déplacement du gros disque on obtient alors que le nombre total de déplacements de disques est minoré par $2 \times (2^{n-1} - 1) + 1$ c'est à dire

par $2^n - 1$. Reste le cas où p est le second piquet. Dans ce cas, il doit y avoir un moment ultérieur m' où on effectue le déplacement vers le troisième piquet (le second ne contient alors que le gros disque). On conclue alors comme dans le cas précédent, en remarquant de plus que les étapes entre m à m' sont une perte de temps.

Correction de l'exercice 4.

1. Le gain maximum en D est la valeur en D plus le maximum entre : le gain maximum en A ; le gain maximum en B et le gain maximum en A. On peut oublier de considérer le gain en A, puisqu'avec des valeurs non négatives sur chaque case, c'est toujours au moins aussi intéressant, partant de A, de passer par B ou C pour aller en D plutôt que d'y aller directement.
2. On considère que les coordonnées sont données à partir de zéro.

```
int robot_cupide(int x, int y){

    /* Cas de base */
    if ( (x == 0) && (y == 0) ) then return Damier[x][y];

    /* Autres cas particuliers */
    if (x == 0) then return Damier[x][y] + robot_cupide(x, y - 1);
    if (y == 0) then return Damier[x][y] + robot_cupide(x - 1, y);

    /* Cas général x, y > 0 */
    return Damier[x][y] + max(robot_cupide(x - 1, y),
                              robot_cupide(x, y - 1));
}
```

3. Un appel à `robot_cupide(3, 3)` entraînera six appels à `robot_cupide(1, 1)`. Partant de la dernière case (Sud-Est) du damier, on peut inscrire, en remontant, sur chaque case du damier le nombre d'appels au calcul du gain sur cette case.

...	...	...	1
...	6	3	1
4	3	2	1
1	1	1	1

On retrouve alors le triangle de Pascal à une rotation près (il faut prendre comme sommet du triangle le coin inférieur droit du tableau). Rien d'étonnant du fait de l'identité binomiale :

$$\binom{n}{p} = \binom{n-1}{p-1} + \binom{n}{p-1}. \tag{1}$$

Le nombre d'appels à Robot-cupide(i, j) fait lors du calcul du gain maximum sur la case (x, y) est

$$\binom{x+y-i-j}{x-i}.$$

Il y a donc un nombre important de répétitions des mêmes appels récursifs.

Une version itérative stockant les gains max dans un nouveau tableau (`gain[n][m]` variable globale) permet d'éviter ces répétitions inutiles.

```
int robot_cupide(int x, int y){
    int i, j;

    gain[0][0] = Damier[0][0];
```

```
    /* Bord Nord */
    for (i = 1; i <= x; i++) {
         gain[i][0] = Damier[i][0] + gain[i - 1][0];
    }

    /* Bord Ouest */
    for (j = 1; j <= y; j++) {
         gain[0][j] = Damier[0][j] + gain[0][j - 1];
    }

    /* Autres cases */
    for (j = 1; j <= y; j++) {
        for (i = 1; i <= x; i++) {
            gain[i][j] = Damier[i][j]
                           + max(gain[i - 1][j], gain[i][j - 1];
        }
    }
    // affiche(x, y); <--- pour afficher...
    return gain[x][y];
}
```

Ce n'était demandé mais on peut chercher à afficher la suite des déplacements effectués par le robot. On peut remarquer que le tableau des gains maximaux, c'est à dire le tableau `gain` après exécution de la fonction précédente (robot itératif), permet de retrouver la case d'ou l'on venait quelle que soit la case où l'on se trouve (parmi les provenances possibles, c'est celle ayant la valeur maximum). Il est donc facile de reconstruire le trajet dans l'ordre inverse. Pour l'avoir dans le bon ordre, on peut utiliser une fonction récursive d'affichage qui montre chaque coup *à la remontée* de la récursion c'est à dire *après* s'être appelée.

Le tableau `gain[n][m]` contient les gains maximaux.

```
void affiche(int i, int j){
  if (i == 0 && j == 0) {
     printf("depart");
     return;
  }
  if (i == 0) {
     affiche(i, j - 1);          // <--| noter l'ordre de ces deux
     printf(", aller à droite"); // <--| instructions. (idem suite)
     return;
  }
  if (j == 0) {
     affiche(i - 1, j);
     printf(", descendre");
     return;
  }
  if (gain[i - 1][j] > gain[i][j - 1]) {
     affiche(i - 1, j);
     printf(", descendre");
  }
  else {
     affiche(i, j - 1);
     printf(", aller à droite");
  }
```

```
}
```