1 Le langage Ocaml

1) a)
```
let mot1 = "Bonjour " in
  let mot2 = "et au revoir" in
  let concat = mot1^mot2 in
  print_string concat
;;
```

b)
```
let a = 1515 in (* Annee de Marignan *)
let b = a*a in
b - 1;;
```

c)
```
let rec a n =
  if n=0 then 1.
  else sin (a (n-1))
;;
```

2) a)
```
let f n = int_of_float (float_of_int n ** 8.) ;;
```

b)
```
let f n =
  let a = n*n in
  let b = a*a in
  b*b
;;
```

2 Récursivité

3)
```
let rec u n =
  if n = 0 then 1
  else 3 * (u (n-1)) - 1
;;
```

4)
```
let rec h n =
  if n=0 then 0.
  else 1. /. (float_of_int n) +. h (n-1)
;;
```

5) On peut utiliser des divisions entières car un coefficient binomial est toujours entier

```
let rec b n k =
  if k>n then 0
  else if k=0 then 1
  else n * b (n-1) (k-1) / k
;;
```

7) Exponentiation rapide (algorithme à retenir...)

a)
```
let rec expo a n =
  if n=0 then 1
  else let k = n/2 in
       let b = expo (a*a) k in
       if n mod 2 == 0 then b
       else a*b
;;
```

b)
```
let rec expo2 a n =
  if n=0 then 1
  else let b = expo a (n/2) in
    if n mod 2 = 0 then b*b
    else a*b*b
;;
```

8) a) Mathématiquement, si A, B, C sont des ensembles et $f : A \to B$ et $g : C \to A$ alors $f \circ g : C \to B$.

b)
```
let rec compiter f n =
  if n=0 then fun x -> x
  else compose f (compiter f (n-1))
;;
```

9)
```
let s n =
  let rec a u k =
    if k=0 then u
    else u + a (3*u-1) (k-1)
  in a 1 n
;;
```

3 Pour aller plus loin...

10) L'échelle de Montgomery

a)
```
let rec nb_chiffres n =
  if n=0 then 0
  else 1 + nb_chiffres (n/2)
;;
```

b)
```
let rec chiffre n i =
  if i=0 then n mod 2
  else chiffre (n/2) (i-1)
;;
```

c) On utilise une fonction auxiliaire récursive `boucle : int -> int -> int -> int` qui permet de remplacer la boucle `for`. Les paramètres correspondent aux valeurs des variables `i, a1, a2`. Puisque la boucle est parcourue pour la valeur `i=0` le cas de base se fait lorsque `i` $= -1$.

```
let montgomery a n =
  let rec boucle i a1 a2 =
    if i = -1 then a1
    else if chiffre n i = 0 then boucle (i-1) (a1*a1) (a1*a2)
    else boucle (i-1) (a1*a2) (a2*a2)
  in boucle (nb_chiffres n - 2) a (a*a)
;;
```

d) Remarque préliminaire : si on note p le nombre de chiffres de n (c'est–à–dire $n = \overline{d_{p-1}d_{p-2}\dots d_0}^2$ cet algorithme n'examine pas la valeur de d_{p-1}. En fait cet algorithme utilise le fait que nécessairement $d_{p-1} = 1$ (on rappelle que $n \in \mathbb{N}^*$).
L'idée de la preuve est de montrer l'invariant :

$$\mathtt{a_1} = \mathtt{a}^{\left(\sum_{k=i}^{p-1} \mathtt{d_k} 2^{k-i}\right)} \text{ et } \mathtt{a_2} = \mathtt{a}.\mathtt{a_1}$$

Remarque culturelle : l'algorithme de l'exponentiation rapide (étant donnés a et n, calculer $b = a^n$) est important en cryptographie car il est exécuté en temps $O(\log(n))$ mais l'opération inverse (étant donnés a et b, retrouver n ; il s'agit du problème du *logarithme discret*) ne peut pas se faire en temps "raisonnable" dans certaines structures algébriques. Cependant, l'algorithme vu à la question 7 ne provoque pas le même nombre de multiplications suivant les chiffres de n, et donc une analyse physique fine de l'activité des circuits électroniques lors de l'exécution du calcul de a^n permet de reconstituer n. L'échelle de Montgomery est un algorithme qui calcule a^n en temps $O(\log(n))$, mais qui nécessite le même nombre de multiplications quelque soient les chiffres de n (à p fixé) : il permet donc en cryptographie de se prémunir d'attaques du logarithme discret par analyse physique de l'activité des circuits électroniques.

11) On utilise une fonction auxiliaire récursive (terminale) `miroir : int -> int -> int` telle que l'appel de `miroir n 0` calcule le nombre formé des mêmes chiffres que `n`, mais lus dans l'autre sens (le paramètre `accu` sert de résultat intermédiaire, il contient le miroir des chiffres déjà parcourus).

```
let palindrome n =
  let rec miroir k accu =
    if k=0 then accu
    else miroir (k/10) (10*accu+(k mod 10))
  in n = miroir n 0
;;
```

12) On va coder un remplissage possible du sac à dos comme un entier entre 0 et $2^n - 1$. En effet à chaque tel nombre t correspond de manière unique une partie de $\{0, \cdots n-1\}$ définie comme l'ensemble des i tel que l'écriture binaire de t possède un 1 en i–ème position.

On définit donc deux fonctions `lourdeur : int -> int` et `butin : int -> int` qui étant donné un remplissage codé par un entier renvoient respectivement la masse totale et un entier donnant la valeur totale du sac.

```
let lourdeur t =
  let rec lourdeurrec t k poidstot =
    if k = n then
```

```
      poidstot
    else if t mod 2 = 0 then
      lourdeurrec (t/2) (k+1) poidstot
    else
      lourdeurrec (t/2) (k+1) (poidstot + poids k)
  in
  lourdeurrec t 0 0
;;

let butin t =
  let rec butin t k =
    if k = n then
      0
    else if t mod 2 = 0 then
      butin (t/2) (k+1)
    else
      (valeur k) + (butin (t/2) (k+1))
  in butin t 0
;;

let sacados () =
  let tmax = (expo1 2 n) in
  let rec parcoursmax t max_courant poids_du_max =
    if t = tmax then
      poids_du_max, max_courant
    else
      let l = lourdeur t in
      if l < poids_max  then
        (* si le sac ne craque pas *)
        let v = butin t in
        if max_courant < v then
          (* si on a trouve meilleur butin *)
          parcoursmax (t+1) v l
        else
          (* si on a pas trouve meilleur butin *)
          parcoursmax (t+1) max_courant poids_du_max
      else
        (* si le sac craque *)
        parcoursmax (t+1) max_courant poids_du_max
  in
  parcoursmax 0 0 0;;
```