1 Alignements

1)
```
let segment_opt x y p q =
  (* on suppose q > p *)
  let n = Array.length x in
  let somme s t i j =
    (* effectue la somme des produits des éléments de s et t *)
    let res = ref 0. in
    for k = i to j do
      res := !res +. s.(k) *. t.(k)
    done;
    !res
  in
  let sx = somme x (Array.make n 1.) p q in
  let sy = somme y (Array.make n 1.) p q in
  let sxy = somme x y p q in
  let sxx = somme x x p q in
  let long = float_of_int (q - p + 1) in
  let a = (long *. sxy -. sx *. sy ) /. (long *. sxx -. sx *. sx) in
  let b = (sy -. a*. sx) /. long in
  a, b
```
et le coût d'un segment
```
let cout_un_segment x y p q =
  (* on suppose q > p *)
  if q = p + 1 then
    (* par deux points passe parfaitement une droite *)
    0.
  else
    let a, b = segment_opt x y p q in
    let res = ref 0. in
    for i = p to q do
      res := !res +.
      (y.(i) -. a *. x.(i) -. b) *. (y.(i) -. a *. x.(i) -. b)
    done;
    !res
```

2) cf 3)

3) La fonction `cout_min` améliorée utilise deux tableaux de taille n, `mintab` qui représente le coût minimum de l'alignement de tous points précédents et `tablopt` qui donne le premier point `p` du segment en cours. Pour chaque point (boucle `A`), on va chercher le début du segment `p` optimal (boucle `B`).

```
let cout_min x y penalite =
  let n = Array.length x in
  let tablopt = Array.make n (0) in
  let mintab = Array.make n infinity in
```

```
  mintab.(0) <- penalite;  (* si il n'y a qu'un ou 2 points *)
  mintab.(1) <- penalite;  (* il y a un unique segment sans erreur *)
  for q = 2 to n - 1 do (* A *)
    let errmin = ref (penalite +. mintab.(q - 1)) in
    (* Initialisation de la recherche du minimum :
       cas où le point est seul sur son segment (p = q) *)
    for p = q - 1 downto 0 do (* B *)
      (* recherche du minimum des couts *)
      let eps = cout_un_segment x y p q in
      let err = penalite +. eps +.
                  if p = 0 then
                  (* cas où le segment revient au début *)
                    0.
                  else
                    mintab.(p - 1) in
      if err < !errmin then
        (errmin := err;
         tablopt.(q) <- p)
    done;
    mintab.(q) <- !errmin
  done;
  mintab, tablopt
;;
```

Et le parcourt de la liste `a` renvoyée pour reconstruire les segments. La fonctions `parcours` affiche le nombre de points de chaque segment en commençant par le dernier.

```
let parcours a =
 let n = Array.length a in
 let i = ref (n - 1) in
 while !i >= 0 do
   let j = a.(!i) - 1 in
   print_int (!i - j);
   print_newline ();
   i := j
 done;
```

4) Enfin la version avec mémoïsation est très similaire :

```
let cout_min_memo x y penalite =
 let n = Array.length x in
 let tab_cout = Array.make n infinity in
 tab_cout.(0) <- penalite;
 tab_cout.(1) <- penalite;
 let rec cout_min_rec q =
   if q < 0 then
     0.
   else if tab_cout.(q) <> infinity then
```

```
      tab_cout.(q)
    else
      let err_min = ref (penalite +. (cout_min_rec (q - 1))) in
      (* Initialisation de la recherche du minimum :
       cas où le point est seul sur son segment (p = q) *)
      for p = q - 1 downto 0 do
        let eps = cout_un_segment x y p q in
        let err = penalite +. eps +. cout_min_rec (p - 1) in
        if err < !err_min then
          err_min := err;
      done;
      tab_cout.(q) <- !err_min;
      !err_min;
  in
  cout_min_rec (n - 1)
```

2 Code Morse

1) `.` 1 ("e")

   `..` 2 ("ee" et "i")

   `...` 4 ("eee", "ei", "ie" et "s")

   `...-` 8 (toutes les précédentes concaténées avec "t", plus toutes les interprétations de `..` concaténées avec "a", plus toutes les interprétations de `.` concaténées avec "u", plus "v")

2) Chaque lettre de la translittération est codée par au moins un symbole morse d'où $|t| \leq |l|$. D'autre part chaque lettre de la translittération est codée par au plus 4 symboles morse d'où $|l|/4 \leq |t|$.

3) m_1 est le nombre d'interprétation d'un symbole or `-` et `.` ont chacun une unique interprétation donc $m_1 = 1$.
   — Soit une translittération du message $l[0] \cdots l[i]$ et soit α la dernière lettre de la translittération que l'on notera $v = w\alpha$. Il y a 4 cas :
   — Si α est codé par un unique symbole, alors w est une translittération du message $l[0] \cdots l[i-1]$,
   — si α est codé sur deux symboles, alors w est une translittération du message $l[0] \cdots l[i-2]$,
   — si α est codé sur trois symboles, alors w est une translittération du message $l[0] \cdots l[i-3]$,
   — si α est codé sur quatre symboles, alors w est une translittération du message $l[0] \cdots l[i-4]$
   On en déduit immédiatement la relation de récurrence

$$m_{i+1} = m_i + m_{i-1} + m_{i-2} + m_{i-3}$$

— Enfin si $i \leq 3$ le même raisonnement est valable sauf que l'on ne considère jamais les m_j pour $j \leq 0$ et d'autre part il y a tous les cas où la translittération se réduit à α (dans ce cas on doit compter toutes les nouvelles lettres). Ainsi pour tous $i \leq 3$

$$m_{i+1} = \left(\sum_{j=1}^{i} m_j\right) + 1$$

4) Le fonction `demorse` peut échouer si le bout de message envoyé ne code pas une lettre, par exemple `..--` dans ce cas cela veut juste dire que l'on ne doit pas modifier `t.(i)` pour cette valeur de `long`. La fonction `demorse` soulève dans ce cas une *exception* (mot clef `raise`) qui est *rattrapée* par la structure `try ... with` et l'on indique avec un filtrage sur les exceptions ce que l'on veut faire dans ce cas. Mais en ignorant ces problèmes, la fonction est simple : on initialise $t[0]$ au mot vide, cela nous permettra d'initialiser `t.(1), t.(2), t.(3)` et `t.(4)` avec la même relation que le cas général.

```
let translitt message =
  let n = String.length message in
  let t = Array.make (n + 1) [] in
  t.(0) <- [ "" ];
  let ajoute_lettre mot =
    let l = String.make 1 lettre in
    mot ^ l
  in
  let rec calculti i =
    if i < n + 1 then begin
      for long = 1 to min 4 i do
         let lettre = demorse (String.sub message (i - long) long)  in
         t.(i) <- (List.map (ajoute_lettre lettre) t.(i - long)) @ t.(i)
      done;
      calculti (i + 1)
      end
  in
  calculti 1;
  t
```