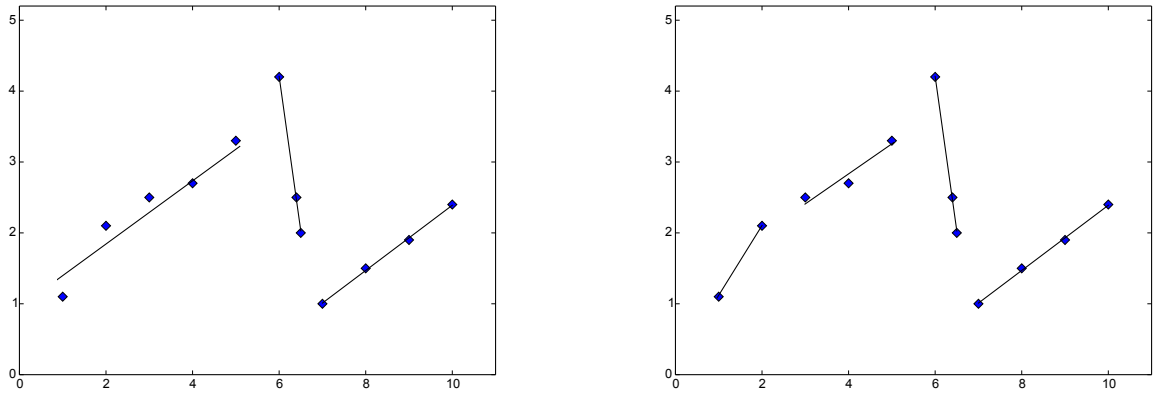
1 Alignements

On dispose d'enregistrements de points de coordonnées successives $(x_0, y_0), \dots, (x_{n-1}, y_{n-1})$, et on cherche une approximation statistique de ces enregistrements par une fonction affine par morceaux. Ces coordonnées sont mémorisées dans deux tableaux `x, y : float array`, tous deux de longueur n. Pour tout $i \in \{0, 1, \cdots n-1\}$, `x[i]` vaut x_i et `y[i]` vaut y_i. Par hypothèse, les valeurs du tableau `x` sont strictement croissantes.
Les points (x_i, y_i) sont supposés être *à peu près* positionnés sur un certain nombre de segments. Le but est de reconstituer ces segments. On supposera de plus que si le $i$ème point est dans un segment mais le $i+1$ème ne l'est pas alors aucun des points qui suivent ne seront sur ce segment.



Alignements optimaux pour une pénalité de 1 et de 0.1 par segments.

On admettra le résultat suivant : soient $0 \leqslant p < q < n$, si les points $(x_p, y_p), \dots, (x_q, y_q)$ sont sur la même droite, la meilleure droite possible approchant ces points est la droite D d'équation $y = ax + b$ avec :

$$a = \frac{(q-p+1)\sum_{i=p}^{q} x_i y_i - \left(\sum_{i=p}^{q} x_i\right)\left(\sum_{i=p}^{q} y_i\right)}{(q-p+1)\sum_{i=p}^{q} x_i^2 - \left(\sum_{i=p}^{q} x_i\right)^2}, \qquad b = \frac{\sum_{i=p}^{q} y_i - a\sum_{i=p}^{q} x_i}{q-p+1}$$

On ne cherchera pas à comprendre ces formules ni à les démontrer, on pourra les utiliser telles quelles. Mathématiquement il s'agit d'une régression linéaire.
On définit alors l'erreur commise en approchant $(x_p, y_p), \dots, (x_q, y_q)$ par la droite D :

$$\varepsilon(D) = \sum_{i=p}^{q} (y_i - ax_i - b)^2$$

On remarque que $\varepsilon(D) \geqslant 0$, et que $\varepsilon(D) = 0$ si et seulement si $\forall i \in [p, q], y_i = ax_i + b$ c'est-à-dire si et seulement si D passe exactement par $(x_p, y_p), \dots, (x_q, y_q)$.
Bien sûr, on pourrait relier les points successifs deux par deux : dans ce cas, chaque segment est une approximation exacte, et l'erreur totale est nulle. Mais cela ne correspond pas à notre approche statistique où l'on cherche à avoir peu de segments.

Pour éviter cela, on définit une pénalité qui est un réel strictement positif, et on rajoute cette pénalité pour chaque nouveau segment. Ainsi, si on approxime nos n points par k segments de droites $D_0, \dots, D_{k-1}$, l'erreur totale commise est :

$$erreur = k \times \text{penalite} + \sum_{\ell=0}^{k-1} \varepsilon(D_\ell)$$

Le but est de déterminer k et $D_0, \dots, D_{k-1}$ de manière à minimiser *erreur*.

1) Écrire une fonction :

   ```
   segment_opt : float array -> float array -> int -> int -> float * float
   ```

   telle que l'appel de `segment_opt x y p q` renvoie le couple (a, b) pour lequel la droite D d'équation $y = ax + b$ approxime au mieux les points $(x_p, y_p), \dots, (x_q, y_q)$.

   Écrire aussi une fonction :

   ```
   cout_un_segment : float array -> float array -> int -> int -> float
   ```

   telle que l'appel de `cout_un_segment x y p q` calcule dans ce cas $\varepsilon(D)$.

2) Supposons qu'on cherche une approximation optimale des points $(x_0, y_0), \dots, (x_q, y_q)$ avec $q < n$. Pour $p \in \{0, 1, \cdots q\}$, on détermine ce que nous coûterait de placer un segment entre p et q. On calcule donc E_{p-1} l'erreur optimale pour les points $(x_0, y_0), \cdots, (x_{p-1}, y_{p-1})$, à laquelle on ajoute $\varepsilon(D_{p,q})$+penalite, à savoir l'erreur associé aux points $(x_p, y_p), \cdots, (x_q, y_q)$. On minimise alors cette somme pour obtenir E_q :

   $$E_q = \min_{p \in [\![0,q]\!]} (E_{p-1} + \varepsilon(D_{p,q}) + \text{penalite})$$

   En utilisant la technique de la programmation dynamique, écrire une fonction : `cout_min : float array -> float array -> float -> float` telle que l'appel à cette fonction `cout_min x y penalite` calcule l'erreur minimum possible.

   Cette fonction devra être entièrement itérative, sans appel récursif, en faisant croître les valeurs successives de q.

   Test : avec

   ```
   x = [|1.0; 2.0; 3.0; 4.0; 5.0; 6.0; 6.4; 6.5; 7.0; 8.0; 9.0; 10.|]
   y = [|1.1; 2.1; 2.5; 2.7; 3.3; 4.2; 2.5; 2.0; 1.0; 1.5; 1.9; 2.4|]
   penalite = 1.0
   ```

   on doit trouver une erreur optimale de l'ordre de 3.176

3) Modifier votre fonction pour qu'elle renvoie l'erreur optimale ainsi que le tableau `antecedent : int array` des antécédents optimaux, c'est-à-dire pour tout q, `antecedent.(q)` est l'indice p du début du dernier segment optimal pour les points $(x_0, y_0), \dots, (x_q, y_q)$. Comment utiliser le tableau `antecedent` pour déterminer les alignements ?

   Test : avec l'exemple précédent, on doit trouver

   ```
   val antecedent : int array = [|0; 0; 0; 0; 0; 0; 5; 5; 6; 6; 8; 8|]
   ```

4) Même question que la question 2 mais cette fois-ci la fonction `cout_min` devra utiliser une fonction auxiliaire récursive. Elle partira de $q = n - 1$ et elle utilisera la technique de la mémoïsation.

2 Code Morse

Le code Morse est composé de points et traits représentant les lettres de l'alphabet. En voici une table de codage

A	. -	B	- . . .	C	- . - .	D	- . .
E	.	F	. . - .	G	- - .	H	. . . .
I	. .	J	. - - -	K	- . -	L	. - . .
M	- -	N	- .	O	- - -	P	. - - .
Q	- - . -	R	. - .	S	. . .	T	-
U	. . -	V	. . . -	W	. - -	X	- . . -
Y	- . - -	Z	- - . .				

Mais comme aucun des espaces n'a été codé il peut y avoir plusieurs interprétations d'un message. Par exemple, la suite `----.-..---.-..` peut être l'une des suite de mots suivants : `OCAML, ON NAMED, OTE DORE, TO LOL, ...` mais également `TMTL MND, TTTTRJRE, OTA IMCE ...`
Un humain peut deviner où le découpage doit être fait grâce à sa connaissance du langage mais pour une machine c'est plus complexe.
Cependant il est possible que plusieurs textes soient des interprétations valides du message.
On s'intéresse à toutes les translittérations (traduction lettre à lettre) possibles d'un message, qu'elles aient du sens ou non. On donne en annexe une fonction `demorse : string -> char` qui correspond au tableau ci-dessus (ex : `demorse "-.."` renvoie `'D' : char`)[1].

1) ✎ Combien il y a-t-il de translittérations de `"."` ? Combien pour `".."`, pour `"..."` et `"...-"` ?

2) ✎ Donner un encadrement de la longueur d'une translittération à l'aide de la longueur du message.

3) ✎ On suppose le message $l = l_0 \cdots l_{n-1}$ fixé de taille n, on note m_i le nombre de translittérations du texte constitué des $i-1$ premiers symboles du message.
   — Que vaut m_1 ?
   — Soit $i > 3$. Exprimer m_{i+1} en fonction de m_i, m_{i-1}, m_{i-2} et m_{i-3}. On pourra remarquer qu'une lettre de l'alphabet latin est codé par au plus 4 symboles et distinguer les cas selon la dernière lettre de la translittération.
   — Que ce passe-t-il si $i \leq 3$?

4) 💻 Écrire une fonction `translitt : string -> string list array` tel que `translitt message` renvoie un tableau `t` où t_i désigne la liste de toutes les translittérations possibles de $l_0, l_1, \cdots l_{i-1}$. On pourra raisonner comme à la question précédente et utiliser la fonction `List.map : ('a -> 'b) -> 'a list -> 'b list` qui applique une fonction à l'ensemble des éléments d'une liste et renvoie la liste des résultats.

1. En cas d'entrée ne codant pas une lettre, ex `..--` la fonction `demorse` renvoie le caractère '_'