MP2I – DS3 2023-2024 – 2h

Pour chaque question, le langage imposé par l'énoncé doit être respecté.
Les fonctions doivent être documentées et commentées à bon escient. Le code doit être indenté et d'un seul tenant (pas de flèche pour dire qu'un bout de code se situe ailleurs par exemple). Le code illisible ne sera pas corrigé.
Sauf précision contraire, les réponses doivent être justifiées.

Tout document interdit. Calculatrice interdite. Téléphone éteint et rangé dans le sac. Pas de montre connectée.

Ce qui est illisible ou trop sale ne sera pas corrigé.
Soulignez ou encadrez vos résultats.

L'énoncé est composé de deux parties. Celles-ci sont indépendantes, mais certaines fonctions de la premières parties peuvent servir dans la deuxième. Vous pouvez toujours utiliser une fonction demandée sans l'avoir écrite.
Tout l'énoncé est à traiter en `OCaml`.

Correction :
De façon générale, et comme je l'ai déjà dit : il faut documenter toute fonction non explicitement demandée. Dès qu'une fonction est un peu compliquée, ajouter des commentaires pour expliciter l'algorithme que vous avez en tête ne peut pas faire de mal. Aucun correcteur ne passera 10 minutes à essayer de comprendre une de vos fonctions, c'est à vous de faire en sorte que ce soit facile à lire. Des fois, il suffit de prendre un nom de fonction explicite plutôt que de prendre `aux`.

Première partie : Exercices – Gymnastique sur les listes

Question 1 : Écrire une fonction `apparait : 'a -> 'a list -> bool` qui teste si un élément appartient à une liste.

Correction :

```
let rec apparait x = function
  | [] -> false
  | h :: t -> h = x || apparait x t
```

Question 2 : On considère la fonction

```
let rec maximum lst =
  match lst with
  | [] -> None
  | x :: suite -> match maximum suite with
                  | None -> Some x
                  | Some m -> Some (max x m)
```

1. Donner les valeurs de `maximum []` et `maximum [1;2;3;4;5]`.
2. Expliquer en le justifiant ce que fait la fonction `maximum`.

Correction :

1. `maximum [] = None` et `maximum [1;2;3;4;5] = Some 5`
   Pour rappel, le typage est fort en `OCaml`*. Il est donc immédiat que les réponses ne peuvent pas être* `None` *(qui est une option) et* `5` *(qui est un entier).*
2. *Ce n'est pas parce qu'il n'est pas écrit "correction totale" que tout à coup ça veut dire autre chose que ça.*
   La valeur de `maximum lst` est une option : `None` si `lst` est vide, et `Some m` où `m` est le maximum de `lst` sinon.
   Montrons-le par récurrence sur la longueur de la liste fournie.

Si la liste `lst` est vide, alors l'appel se termine en ligne 3 et on a bien `maximum lst = None`.
Supposons que tout appel à `maximum lst` se termine et est correct pour une liste `lst` quelconque de longueur n pour un certain entier $n \geq 0$.
Soit une liste `lst'` de longueur $n+1$. L'appel `maximum lst'` passe par la règle de la ligne 4 : `x :: suite = lst'`, donc `suite` est de longueur n. L'appel récursif `maximum suite` se termine donc et est correct par hypothèse de récurrence. Le filtrage de cette expression donne la valeur recherchée :

- si `suite = []`, alors l'expression évaluée est celle de la ligne 5, soit `Some x`, qui est correcte.
- sinon, alors `maximum suite = Some m` où `m` est le maximum de `suite` par hypothèse de récurrence, et `max x m` est bien le maximum de `lst'`.

Question 3 : Écrire une fonction `map : ('a -> 'b)-> 'a list -> 'b list` qui applique une fonction à tous les éléments d'une liste.
exemples d'utilisation :

```
utop # map (fun x -> x - 2) [1; 2; 3; 4; 5];;
- : int list = [-1; 0; 1; 2; 3]
utop # map List.length [[]; [3; 2; 1]; [1]; [9; 3]];;
- : int list = [0; 3; 1; 2]
```

Correction :

```
1 let rec map f = function
2   | [] -> []
3   | x :: xs -> f x :: map f xs
```

Question 4 : Écrire une fonction `trier : 'a list -> ('a -> 'a -> bool)-> 'a list` dont le deuxième argument est une fonction qui permet de comparer deux valeurs de type `'a` : appliquée à des arguments `a` et `b`, cette fonction renvoie `true` si et seulement si `a` est considéré comme inférieur à `b`.
Vous pouvez choisir l'algorithme de tri que vous voulez, mais il faut donner son nom ou expliquer son principe.
exemples d'utilisation :

```
utop # trier [3;2;1;5;-4] (fun x y -> x < y);;
- : int list = [-4; 1; 2; 3; 5]
utop # trier [3;2;1;5;-4] (fun x y -> x > y);;
- : int list = [5; 3; 2; 1; -4]
```

Correction :
On l'a fait en TD pour l'ordre naturel sur les entiers et c'était à rendre en DM. Si vous ne savez pas refaire ce qui a déjà été fait, il faut vous poser des questions sur vos méthodes de travail. La base c'est le cours, la couche du dessus ce qui a été fait ensemble. Vous ne pourrez pas résoudre efficacement (=correctement et rapidement) de nouveaux exercices si vous ne savez pas résoudre des exercices déjà traités.

```
1 (* inf x y vaut vrai si et seulement si x possède une valeure inférieure ou égale
      à y *)
2 let rec trier lst inf = (* trie dans l'ordre croissant selon inf *)
3   let rec inserer e = function (* liste supposée triée pour l'ordre induit par
        inf *)
4     | [] -> [e]
5     | x :: xs -> if inf e x then e :: x :: xs else x :: inserer e xs
6   in match lst with
7      | [] -> []
8      | x :: xs -> inserer x (trier xs inf)
```

DEUXIÈME PARTIE : PROBLÈME – SPÉLÉO-LOGIQUE

Correction :
Ce qui suit a été emprunté à Marie Durand, professeure d'informatique au lycée Champollion, puis adapté. Le sujet est inspiré du sujet d'Informatique Tronc Commun X-ENS 2022.

Les questions dont le numéro est marqué d'un étoile (comme la question 7) sont difficiles, ce marquage n'est pas là pour vous empêcher de les faire, mais pour vous aider à apprendre à gérer votre temps.

Le problème
Nous allons déterminer le remplissage d'une grotte lors d'une inondation alimentée par une source d'eau localisée quelque part dans la grotte. La grotte considérée sera bi-dimensionnelle et décrite par le profil de son fond. On suppose défini le type `direction` suivant :

```
type direction = B | H | D | G
```

représentant les directions verticales (`B` : bas, `H` : haut) et horizontales (`G` : gauche, `D` : droite).

Question 5 : Écrire une fonction `oppose : direction -> direction` qui prend une direction en argument et renvoie la direction opposée (par exemple `oppose B` vaut `H`).

Correction :

```
let oppose = function
  | B -> H
  | H -> B
  | D -> G
  | G -> D
```

Il faut privilégier les filtrages par motifs sur la suite de `if ... then ... else ...` imbriqués en `OCaml`, sauf quand on teste une expression booléenne. C'est plus idiomatique et sur machine ça permet de profiter des vérifications faites par le compilateur.

Le profil de la grotte sera donné sous la forme d'une suite de pas horizontaux ou verticaux de longueur 1, encodée sous la forme d'une liste composée des constantes `H`, `B`, `G` et `D` pour les quatre directions Haut, Bas, Gauche et Droite. L'origine du profil sera toujours le point `(0, 0)`. On considérera toujours que le profil de la grotte se prolonge à gauche et à droite par deux murs verticaux infinis (en particulier, c'est comme s'il y avait un mur de direction `B` avant le point $(0,0)$ et un mur de direction `H` après le dernier point du profil, mais ces deux murs ne sont pas explicitement écrits dans le profil). Dans tout le sujet, on suppose également que le profil contient toujours au moins un pas `D` vers la Droite (en particulier un profil n'est donc pas vide). Un profil est représenté en `OCaml` par une liste de directions. La figure 1 donne l'exemple d'une grotte et de son profil.

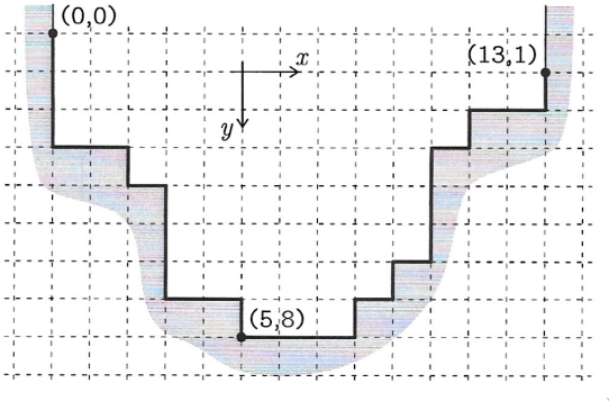
1 Validité d'un profil

On dira qu'un profil est *sans rebroussement* s'il ne contient pas de pas qui revienne immédiatement sur le pas précédent, par exemple pas de `..., G, D, ...`. En particulier, et étant donné les conditions aux bords, le profil d'une grotte sans rebroussement ne commence pas par `H` et ne finit pas par `B`.

Question 6 : Écrire une fonction `est_sans_rebroussement : direction list -> bool` qui teste si le profil est sans rebroussement.

Correction :
Bien lire l'énoncé : c'est écrit deux lignes au-dessus : il y a une condition de début et de fin sur un profil sans rebroussement.

```
let est_sans_rebroussement profil = match profil with
  | [] -> true
```




```
[B; B; B; D; D; B; D; B; B; B; D; D; B; D;
D; D; H; D; H; D; H; H; H; D; H; D; D; H]
```

FIGURE 1 – Une grotte et son profil.

```
3     | H :: _ -> false
4     | _ ->
5        let rec pas_de_retour = function
6          | [] -> failwith "on ne peut pas passer par la"
7          | [ B ] -> false
8          | [ _ ] -> true
9          | x :: y :: lst -> x <> oppose y && pas_de_retour (y :: lst)
10       in pas_de_retour profil
```

Une *vallée* est une grotte dont le profil est sans rebroussement et commence par descendre en ne faisant que des pas vers le Bas ou la Droite, puis remonte en ne faisant que des pas vers le Haut ou la Droite jusqu'à son point d'arrivée (la direction Gauche est en particulier interdite). La grotte de la figure 1 est une vallée.

Question 7* : Écrire une fonction `est_une_vallee : direction list -> bool` qui teste si un profil est une vallée.

Correction :
Soit vous prenez des noms de fonctions auxiliaires clairs, soit vous documentez, mais il faut à tout pris éviter les aux, aux2, etc. dont on ne comprend pas ce qu'ils font.

```
1  let est_une_vallee profil =
2    let rec descend profil =
3      match profil with
4      | [] -> true
5      | (B | D) :: suite -> descend suite
6      | G :: _ -> false
7      | H :: _ -> monte profil
8    and monte profil =
9      match profil with
10     | [] -> true
11     | (B | G) :: _ ->false
12     | (H | D) :: suite -> monte suite
13   in est_sans_rebroussement profil && descend profil
```

On considère désormais que les axes des x et des y pointent respectivement vers la Droite et le Bas. On rappelle que le profil d'une grotte a pour origine la position `(0, 0)`.

Question 8 : Écrire une fonction `voisin : int -> int -> direction -> int * int` qui calcule les coordonnées d'un voisin d'un point dans un certaine direction.
exemple d'utilisation :

```
utop # voisin 3 4 B;;
- : int * int = (3, 5)
utop # voisin 10 2 D;;
- : int * int = (11, 2)
```

Correction :

```
let voisin x y dir = match dir with
  | B -> x, y+1
  | H -> x, y-1
  | G -> x-1, y
  | D -> x+1, y
```

Question 9 : Écrire une fonction `liste_des_points : direction list -> (int * int)list` qui renvoie la liste des coordonnées de tous les points d'un profil, dans l'ordre, de l'origine $(0,0)$ à l'arrivée du profil.
exemple sur le profil de l'énoncé :

```
utop # liste_des_points ex_enonce;;
- : (int * int) list =
[(0, 0); (0, 1); (0, 2); (0, 3); (1, 3); (2, 3); (2, 4); (3, 4); (3, 5);
 (3, 6); (3, 7); (4, 7); (5, 7); (5, 8); (6, 8); (7, 8); (8, 8); (8, 7);
 (9, 7); (9, 6); (10, 6); (10, 5); (10, 4); (10, 3); (11, 3); (11, 2);
 (12, 2); (13, 2); (13, 1)]
```

Correction :
La question était sans ambiguïté, mais l'exemple erroné (apparemment j'ai oublié la dernière compilation), car il ne contenait pas le point (0,0). J'ai neutralisé les points de notation concernant ce premier point du profil, mais j'ai marqué en commentaire l'oubli.

```
let liste_des_points profil =
  let rec points profil x y =
    match profil with
    | [] -> [ (x, y) ]
    | dir :: suite -> let vx, vy = voisin x y dir in
                      (x, y) :: points suite vx vy
  in points profil 0 0
```

La fonction `points : direction list -> int -> int -> (int * int)list` est telle que `points d x y` est la liste des points en partant de `(x, y)` et en suivant la liste de directions `d`.

On dira qu'un profil est *simple* s'il ne repasse pas par le même point.

Question 10* : Écrire une fonction `est_simple : direction list -> bool` qui teste si un profil est simple.

Correction :
La fonction `apparait` a été écrite dans un exercice précédent : vous pouvez la supposer écrite, mais pensez à spécifier que c'est celle-là que vous utilisez. (Si c'est dans le même exercice ce n'est pas la peine de le dire.)

```
let est_simple profil =
  let points = liste_des_points profil in
  let rec apparait p = function
    | [] -> false
    | q :: suite -> p = q || apparait p suite in
  let rec simple = function
    | [] -> true
    | p :: suite -> not (apparait p suite) && simple suite in
  simple points
```

La fonction `simple : 'a list -> bool` teste si la liste fournie en argument comporte plusieurs fois la même valeur ou pas.

Question 11 : Donner en le justifiant l'ordre de grandeur exact de la complexité dans le pire des cas de votre fonction `est_simple`.
Suivant ce que vous trouvez, justifiez brièvement qu'on ne peut pas faire mieux pour résoudre ce problème, ou au contraire donnez une idée rapide de comment on pourrait obtenir une meilleure complexité (aucun code n'est demandé dans cette question).

Correction :
Si la fonction n'a pas été écrite, vous ne pouvez pas calculer sa complexité. Sautez la question dans ce cas.
La complexité temporelle dans le pire des cas de la fonction `apparait` est en $\Theta(n)$ où n est la longueur de la liste fournie en argument. En effet, si on note $T(n)$ le nombre d'opérations élémentaires pour une liste de longueur n, on a à cause de la ligne 5 : $T(n) \leq \alpha + T(n-1)$ pour un certain $\alpha > 0$, donc $T(n) = O(n)$. Par ailleurs cette complexité est atteinte si l'élément ne se trouve pas dans la liste, donc $T(n) = \Theta(n)$.
La complexité temporelle dans le pire des cas de la fonction `simple` est en $\Theta(n^2)$ où n est la longueur de la liste fournie en argument. En effet, si on note $U(n)$ le nombre d'opérations élémentaires pour une liste de longueur n, on a à cause de la ligne 8 :

$$U(n) \leq \underbrace{\alpha n}_{\text{appel à } \texttt{apparait}} + U(n-1)$$

pour un certain $\alpha > 0$, donc $U(n) = O(n^2)$ par somme télescopique :

$$\begin{aligned} U(n) &\leq \alpha n + U(n-1) \\ U(n-1) &\leq \alpha(n-1) + U(n-2) \\ &\dots \\ U(1) &\leq \alpha + U(0) \\ \hline U(n) &\leq \alpha \sum_{k=1}^{n} k = \alpha \frac{n(n+1)}{2} \end{aligned}$$

Par ailleurs cette complexité est atteinte s'il n'y a pas de doublon dans la liste, donc $U(n) = \Theta(n^2)$.

2 Vallée

Dans cette partie, nous considérerons que les profils sont toujours de type *vallée*, sans avoir à le vérifier.
Le *fond* d'une vallée est son point le plus à gauche parmi ses points les plus bas. Le fond de la vallée de la figure 2 a pour coordonnées `(5, 8)`.

Question 12* : Écrire une fonction `fond : direction list -> int * int` qui calcule les coordonnées du fond de la vallée fournie en profil.

Correction :
Deux approches possibles : prendre la liste des points et chercher le bon dedans, ou simuler le parcours du début à la fin. J'ai choisi la 2e solution.

```
1  let fond profil =
2    if not (est_une_vallee profil) then failwith "pas une vallee"
3    else
4      let rec le_fond g (x, y) = function
5        (* g : point le plus à gauche au niveau actuel
6           (x, y) : point actuel *)
7        | [] | H :: _ -> g
8        | D :: suite -> le_fond g (voisin x y D) suite
9        | B :: suite -> let g = voisin x y B in le_fond g g suite
10     in le_fond (0, 0) (0, 0) profil
```

On considère à présent qu'au temps $t = 0$, une source d'eau *située au fond de la vallée* commence à couler avec un *débit constant* et à remplir la vallée. L'objectif de cette partie est de calculer quelle sera la hauteur de l'eau dans la vallée à chaque instant t. On considérera que *le débit de la source est unitaire*, c'est-à-dire d'une unité de surface (un carreau) par unité de temps. La figure 2 indique le niveau de l'eau à différentes dates t dans la vallée de la figure 1.
On appelle *plateau* tout segment horizontal *maximal* du profil de la vallée. Un plateau est défini par le triplet (x_0, x_1, y) où $x_0 < x_1$ sont les abscisses de ses deux extrémités et y est leur ordonnée. La vallée de la figure 2 possède exactement 8 plateaux, indiqués en gras sur la figure : `(0, 2, 3)`, `(2, 3, 4)`, `(3, 5, 7)`, `(5, 8, 8)`, `(8, 9, 7)`, `(9, 10, 6)`, `(10, 11, 3)`, `(11, 13, 2)`.

Question 13* : Écrire une fonction `plateaux : direction list -> (int * int * int)list`, de complexité temporelle linéaire en la longueur du profil (qui doit être une vallée) qui renvoie la liste des triplets correspondant à ses plateaux (pas d'ordre imposé sur cette liste). Justifier brièvement la complexité.

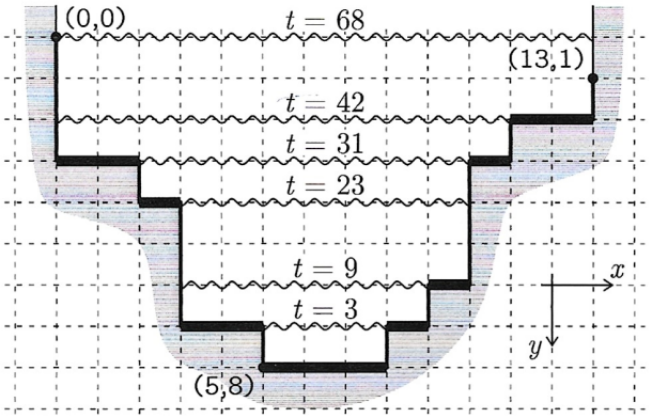


FIGURE 2 – Remplissage d'une vallée

Correction :
Je la laisse sans commentaire, pour que vous voyez comme c'est compliqué à lire. Et encore mon indentation est correcte et il n'y a pas des ratures régulièrement réparties dans le code.

```
let plateaux profil =
  if not (est_une_vallee profil) then failwith "pas une vallee"
  else
    let rec pla vallee ppp xg xd y =
      match vallee with
      | [] -> if xg<>xd then (xg, xd, y) :: ppp else ppp
      | G :: _ -> failwith "pas une vallee"
      | D :: suite -> pla suite ppp xg (xd+1) y
      | H :: suite -> let ppp = if xg <> xd then (xg, xd, y) :: ppp else ppp in
                      pla suite ppp xd xd (y-1)
      | B :: suite -> let ppp = if xg <> xd then (xg, xd, y) :: ppp else ppp in
                      pla suite ppp xd xd (y+1)
    in pla profil [] 0 0 0
```

Remarquons que si l'on trie les plateaux d'une vallée du plus profond au moins profond (par y décroissants), on obtient une décomposition du volume intérieur de la vallée en rectangles. Ces rectangles sont délimités verticalement par les ordonnéees consécutives des plateaux et horizontalement par les abscisses des extrémités des plateaux et/ou le bord de la grotte. La vitesse de montée des eaux est constante à l'intérieur de chaque rectangle et vaut exactement $1/w$ où w est la largeur du rectangle. L'eau met donc un temps $h\,w$ à remplir chaque rectangle de taille $w \times h$. Dans le cas de la vallée illustrée ci-dessus, la liste des tailles (w,h) des rectangles obtenus est, de bas en haut : `[(3, 1); (6, 1); (7, 2); (8, 1); (11, 1); (13, -1)]` où la valeur `-1` de la dernière hauteur indique par convention que le dernier rectangle est de hauteur infinie.
On donne la fonction suivante pour obtenir cette liste de rectangles :

```
let rectangles plateaux =
  let plat_tries = trier plateaux (fun (x1,_,y1) (x2,_,y2) -> y1 > y2 || (y1 = y2 &&
      x1 < x2)) in
  match plat_tries with
  | [] -> []
  | (x1, x2, y) :: suite ->
     let rec rect deb fin haut plat =
       match plat with
       | [] -> [(fin-deb, -1)]
       | (x1, x2, y) :: (a1, a2, b) :: suite when y = b ->
          (fin-deb, haut-y) :: rect x1 a2 y suite
       | (x1, x2, y) :: suite ->
          let r = (fin-deb, haut-y) in
          if x2 > fin
          then r :: rect deb x2 y suite
          else r :: rect x1 fin y suite
```

```
16      in rect x1 x2 y suite
```

où la fonction `trier` est celle que vous deviez écrire à la question 4.

Question 14 : Que vaut `plat_tries` (défini en ligne 2) si le paramètre `plateaux` contient la liste des plateaux de la figure 2 ?

Correction :
On a :

```
1 plat_tries = [(5, 8, 8); (3, 5, 7); (8, 9, 7); (9, 10, 6); (2, 3, 4); (0, 2, 3);
2   (10, 11, 3); (11, 13, 2)]
```

Question 15 : Montrer que la complexité temporelle dans le pire des cas de la fonction locale `rect` (définie à partir de la ligne 6) est au plus linéaire en la longueur de son dernier paramètre.

Correction :
Ici on n'a pas besoin de comprendre la fonction pour calculer sa complexité (et c'était fait exprès). Si `plat` est une liste vide, l'appel se termine en temps constant à la ligne 8. Sinon, si la liste est de longueur $n > 0$, il y a un appel récursif :

- soit en ligne 10 sur une liste de longueur $n-2$,
- soit à une des lignes 14 ou 15 sur une liste de longueur $n-1$.

Le nombre $T(n)$ d'opérations élémentaires effectuées par un appel à `rect` sur une liste de longueur n vérifie donc $T(n) \leq \alpha + T(n-1)$ pour une certaine constante α. Donc $T(n) = O(n)$.

Question 16 : Écrire une fonction `aires_rectangles : direction list -> int list` qui renvoie la liste des aires des rectangles d'une vallée, triée de bas en haut (en fonction des positions des rectangles dans le schéma), sauf pour le dernier rectangle (qui est infini) pour lequel on veut l'opposé de sa largeur.
exemple d'utilisation sur la grotte de la figure 1 :

```
utop # aires_rectangles ex_enonce;;
- : int list = [3; 6; 14; 8; 11; -13]
```

Correction :
En utilisant la fonction `map` de l'exercice 3 :

```
1 let aires_rectangles profil =
2   map (fun (x, y) -> x * y) (rectangles (plateaux profil))
```

Question 17 : Écrire une fonction `hauteur_de_l_eau : float -> direction list -> float` qui prend en argument un temps de remplissage (positif) et le profil d'une valléee et renvoie la hauteur de l'eau (mesurée depuis le fond) dans la vallée au bout de ce temps.
exemple d'utilisation :

```
utop # hauteur_de_l_eau 16. ex_enonce;;
- : float = 3.
utop # hauteur_de_l_eau 25. ex_enonce;;
- : float = 4.25
```

Correction :
Petit piège dans cette question car il ne suffit pas de connaître les aires des rectangles pour répondre, il ne fallait donc pas utiliser la question précédente. Par ailleurs il faut faire très attention aux conversions.

```
1 let hauteur_de_l_eau t profil =
2   let rects = rectangles (plateaux profil) in
3   let rec temps t = function
4     | [] -> failwith "impossible"
5     | [(x, -1)] -> t /. float_of_int x
6     | (x, y) :: suite -> if t > float_of_int (x*y) then
7                          let tt = float_of_int (x*y) in float_of_int y +. temps (t
```

```
                            -. tt) suite
8                           else t /. float_of_int x
9   in temps t rects
```