I. Il s'agit de la fonction de Mc Carthy. Nous allons montrer que le calcul de `f (x)` termine toujours sur $\mathbb{Z}$ et vaut : $x-10$ si $x>100$ et 91 si $x\leq 100$.

L'intervalle d'entiers $]\!]100,+\infty[\![$ va constituer notre ensemble $\mathcal{B}$ de cas de base ; nous allons en effet, pour $x\leq 100$, procéder par induction sur $(]\!]-\infty,100]\!],\geq)$ qui est bien fondé (car majoré).

Nous allons définir le prédicat suivant :
$\mathfrak{p}_f(x)$ = « $f(x)$ termine et
« - si $100<x$, $f(x)=x-10$,
« - si $x\leq 100$, $f(x)=91$ »
et le montrer en suivant le théorème de correction (attention, l'ordre considéré est l'opposé de l'ordre naturel sur les entiers ; notre preuve inductive va donc aller *a contrario* de nos habitudes de preuve par récurrence). Trois cas se présentent :

- soit $x>100$, $f(x)$ termine et vaut bien $x-10$, c'est un cas de base.

- soit $90\leq x\leq 100$ et par définition $f(x)=f(f(x+11))$. Comme $101\leq x+11$, $f(x+11)=x+11-10=x+1$ et donc dans cet intervalle $f(x)=f(x+1)$. On peut recommencer ce raisonnement sur $x+1$, puis $x+2$... tant que l'on reste dans l'intervalle $[\![90,100]\!]$. On obtient alors $f(x)=f(x+1)=\ldots=f(100)=f(101)$. Car $f(101)$ constitue le cas d'arrêt rencontré. La valeur calculée vaut donc $f(101)=91$. Le prédicat est donc à nouveau vérifié dans ce cas.

- soit $x<90$ et $f(x)$ conduit encore au calcul de $f(f(x+11))$. Comme $101\geq x+11>x$, par hypothèse d'induction $f(x+11)$ termine et vaut $91>x$. Donc $f(x)$ termine et vaut $f(f(x+11))=f(91)=91$. Le prédicat est donc à nouveau vérifié dans ce cas.

Finalement, on a bien montré, en suivant le théorème de correction, le résultat annoncé.

Différentes façons de trier...

II. *Le tri par sélection.*

1° On peut proposer :

```
let rec minimum_et_reste = function
    | [x] ->(x,[])
    | x1::r1 -> let (m2,l2) = (minimum_et_reste r1) in
                if x1<m2 then (x1,m2::l2) else (m2,x1::l2);;
```

2° Si la liste à trier est de longueur n, l'algorithme applique successivement `minimum_et_reste` à la liste initiale, la liste privée de son minimum, celle-ci privée de son minimum... jusqu'à la liste vide, donc à des listes de longueur n, $n-1$, $n-2$,..., 0. Pour déterminer le minimum d'une liste `l`, la fonction `minimum_et_reste` parcourt entièrement `l` et effectue (longueur `l`) -1 comparaisons. L'algorithme de tri par sélection effectue donc en tout $\dfrac{n(n-1)}{2}$ comparaisons.

III. *Le tri par insertion*

1° a. On propose :

```
let rec insere element = function
    | [] -> [element]
    | x::reste -> if element <= x
                  then element::x::reste
                  else x::(insere element reste);;
```

b. On a alors :

```
let rec tri_insertion = function
    | [] -> []
    | x::reste -> insere x (tri_insertion reste);;
```

2° L'insertion dans une liste déjà triée n'impose pas son parcours complet contrairement à une recherche de minimum. Le tri par insertion devrait donc mieux se comporter que le tri par sélection.

a. Sauf erreur de ma part, il y en a 21.

b. La somme du nombre d'inversions d'une permutation et de sa permutation miroir est $\frac{n(n-1)}{2}$, le nombre de couples possibles. En regroupant deux par deux les permutations, le nombre moyen d'inversions d'une permutation de $\mathfrak{S}_n$ est donc $\frac{n(n-1)}{4}$.

c. Le nombre de comparaisons effectuées par `insere` pour insérer l'élément l_i dans le bout de liste déjà trié correspond au nombre d'inversions présentes dans la suite des éléments d'indice i à n de la liste initiale, plus 1 (le premier test). Pour une permutation σ donnée, le nombre total de comparaisons effectuées est donc :

$$c_\sigma = n - 1 + inv(\sigma)$$

Le nombre moyen de comparaisons effectuées par le tri par insertion est donc :

$$\frac{1}{n!}\sum_{\sigma\in\mathfrak{S}_n} c_\sigma = n - 1 + \frac{n(n-1)}{4} = \frac{n(n+3)}{4} - 1$$

soit encore un nouvel algorithme quadratique, plus efficace en pratique que le tri par sélection.

3° Dans le cas où la liste initiale est « presque en ordre », le nombre de comparaisons effectuées est faible, ce qui n'est pas le cas du tri par sélection.

IV. *Le tri bulle.*

1° On peut proposer :

```
let rec une_passe  = function
  | [(x:int)] -> false,[x]
  | x::reste  -> let bool,res = (une_passe reste) in
                 if x<=(hd res)
                 then bool,x::res
                 else true,(hd res)::x::(tl res);;

let rec tri_bulle  = function
  | [] -> []
  | l -> let (modifiee, liste) = une_passe l in
          if modifiee
          then (hd liste)::(tri_bulle (tl liste))
          else liste;;
```

2° **a.** Comme dans le tri par sélection, l'algorithme effectue successivement `une_passe` sur la liste initiale, la liste privée de son minimum, celle-ci privée de son minimum... jusqu'à tomber sur une liste triée (qui au pire est de longueur 1). Dans ce cas, le parcours complet des listes successives impose d'effectuer $(n-1)+(n-2)+(n-3)+\cdots+1$ comparaisons. Le nombre total de comparaisons est donc bien majoré par $\frac{n(n-1)}{2}$.

b. Chaque échange d'éléments de tri bulle supprime une et une seule inversion et, une fois le tri terminé, il n'y a plus aucune inversion. Ainsi le nombre total d'échanges effectués est égal au nombre d'inversions dans la liste initiale. Le nombre moyen d'échanges effectués par tri bulle est donc le nombre moyen d'inversions dans $\mathfrak{S}_n$ (que nous avons déjà calculé) soit $\frac{n(n-1)}{4}$.

V. *Le tri fusion.*

1° On peut proposer :

```
let rec divise = function
    | [] -> ([],[])
    | [e] -> ([e],[])
    | a::b::r -> let (m1,m2) = divise r in
                 (a::m1,b::m2);;
let rec fusion = fun
    | l [] -> l
    | [] l -> l
    | (a::r as l1) (b::s as l2) -> if a<b
                                  then a::(fusion r l2)
                                  else b::(fusion l1 s);;
```

```
let rec tri_fusion = fun
    | [] -> []
    | [e] -> [e]
    | l -> let (m1,m2) = divise l in
           fusion (tri_fusion m1) (tri_fusion m2);;
```

2° La division d'une liste de longueur n en deux moitiés ne nécessite aucune comparaison, leur fusion en requiert $2\lfloor n/2 \rfloor$. La définition de `tri_fusion` montre que :

$$c(n) = c(n) = c\big(\lfloor n/2 \rfloor\big) + c\big(\lceil n/2 \rceil\big) + 2\lfloor n/2 \rfloor$$

et cette récurrence entraîne que $c(n) = \Theta(n \lg n)$ (vous souvenez-vous des notations Ω, $\mathcal{O}$ et Θ dont j'ai parlé en cours? Dans le cas contraire, je les ré-expliquerai et reprendrai ce calcul...)

3° Cette dernière complexité rend le `tri_fusion` plus intéressant que les autres algorithmes de tri (y compris plus intéressant que le tri rapide vu en cours qui reste quadratique dans le pire des cas).