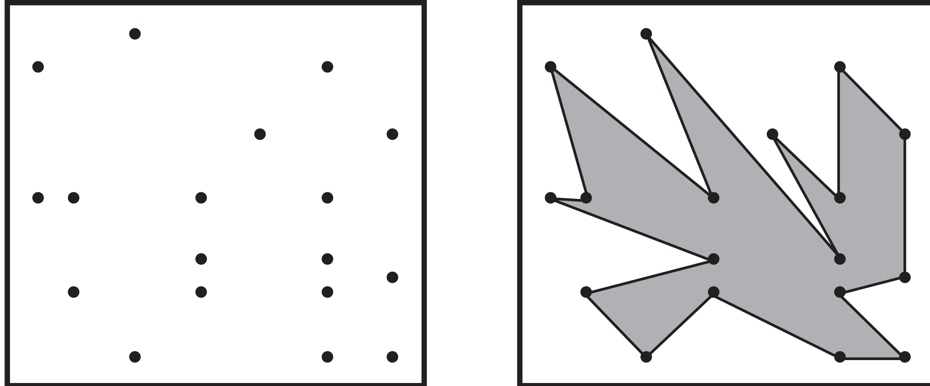
Le problème consiste à trouver un chemin dans un ensemble de n points donnés du plan, chemin qui ne se coupe pas, qui passe par tous les points et qui revient à son point de départ : *un chemin fermé simple.*



Une façon élémentaire de résoudre ce problème est la suivante : choisir un point qui servira d'origine de l'axe polaire (que l'on choisit horizontal par exemple), calculer l'angle (le θ) des coordonnées polaires des autres points, puis trier les points suivant cet angle et enfin relier les points adjacents.
Ce problème va donc être le prétexte à la révision des algorithmes de tri classique sur des listes...

1. *Tri rapide.* Le principe de cet algorithme basique et rapide est le suivant :
- si la liste d'entiers est vide ou réduite à un élément, l'algorithme laisse la liste inchangée,
- si la liste, notée `l`, est de longueur supérieure ou égale à deux, on considère son élément de tête, appelons-le `e`, on sépare la queue de `l` en deux sous-listes : celle des éléments plus grands que `e` et celle des éléments plus petits. Puis on trie, suivant ce même principe, chacune de ces deux sous-listes et enfin on assemble le tout pour former la liste finale triée.

Par exemple, si la liste initiale est la suivante :

```
[18;3;10;25;9;3;11;13;23;8]
```

la liste des éléments inférieurs à l'entier en tête `18` est `[3;10;9;3;11;13;8]`, celle des entiers supérieurs est `[25;23]`. Le tri de ces deux listes fournit `[3;3;8;9;10;11;13]` et `[23;25]`. Le résultat est alors obtenu en mettant bout à bout la première liste triée suivie de `18` suivi de la deuxième liste triée soit `[3 ;3 ;8 ;9 ;10 ;11 ;13 ;`**`18`**` ;23 ;25]`.

Proposer une implémentation en Caml de cet algorithme sous la forme d'une fonction `tri_rapide` qui appelle une fonction `partition` qui réalise la séparation en deux sous-listes en fonction de l'élément de tête.

2. *Le tri par sélection.*
C'est sans doute la méthode de tri la plus élémentaire. Elle consiste à trouver le minimum de la liste (qui peut ne pas être unique), à le placer en tête de la liste résultat et à recommencer cette opération sur les éléments restants.

1° Proposer une implémentation de la fonction `minimum_et_reste` qui isole le minimum d'une liste des autres éléments :

```
#minimum_et_reste [5;4;8;2;6;7;7;2;9];;
- : int * int list = 2, [5; 4; 8; 2; 6; 7; 7; 9]
```

Votre fonction `minimum_et_reste l` *parcourra entièrement* `l` *et effectuera (longueur* `l`*)* -1 *comparaisons.*

2° En déduire la fonction principale `tri_selection`.

3. *Le tri par insertion*
C'est le tri du joueur de cartes. On considère en quelque sorte que les éléments de la liste à trier sont donnés un par un. Le premier élément constitue à lui tout seul une liste de longueur 1 triée. On range alors à sa bonne place le deuxième élément pour former une liste de longueur 2 triée et on continue. On insère ainsi les éléments successivement dans des sous-listes triées.
Par exemple, sur la liste initiale `[5;4;8;2;6;7;7;2;9]`, les étapes de tri successives sont :

```
J'ins\`{e}re 9 dans  []
         ce qui donne [9]
J'ins\`{e}re 2 dans  [9]
         ce qui donne [2; 9]
J'ins\`{e}re 7 dans  [2; 9]
         ce qui donne [2; 7; 9]
J'ins\`{e}re 7 dans  [2; 7; 9]
         ce qui donne [2; 7; 7; 9]
J'ins\`{e}re 6 dans  [2; 7; 7; 9]
         ce qui donne [2; 6; 7; 7; 9]
J'ins\`{e}re 2 dans  [2; 6; 7; 7; 9]
         ce qui donne [2; 2; 6; 7; 7; 9]
J'ins\`{e}re 8 dans  [2; 2; 6; 7; 7; 9]
         ce qui donne [2; 2; 6; 7; 7; 8; 9]
J'ins\`{e}re 4 dans  [2; 2; 6; 7; 7; 8; 9]
         ce qui donne [2; 2; 4; 6; 7; 7; 8; 9]
J'ins\`{e}re 5 dans  [2; 2; 4; 6; 7; 7; 8; 9]
         ce qui donne  [2; 2; 4; 5; 6; 7; 7; 8; 9]
```

(la récursion fait que l'on considère la liste par la fin).

1° Commencez par écrire la fonction `insere e l` qui insère un élément à sa place dans une liste *déjà triée* `l`.

2° En déduire une implémentation récursive du tri par insertion.

4. *Le tri bulle.*
Une variante du tri par sélection est le *tri bulle*. Son principe est de parcourir la liste à trier et de transposer toute inversion rencontrée (une inversion est un couple (i, j) tel que $i < j$ et $\sigma(i) > \sigma(j)$.). À la fin de cette passe, le minimum de la liste se retrouve en tête (si la suite de transpositions s'est faite, en raison de la récursion, en commençant par la fin). On recommence alors l'opération sur la queue de la liste.
Si l'on se représente une liste comme une « colonne verticale de liquide » (la tête de la liste en haut), les nombres les plus légers (les plus petits) remontent tels des bulles vers le haut en poussant des bulles successives du bas vers le haut : c'est l'origine du nom « tri bulle ».
Dans le détail, les passes successives du tri bulle sont, par exemple sur la liste initiale `l = [5;4;8;2;6;7;7;2;9]` :

```
On effectue une passe sur la liste : [5; 4; 8; 2; 6; 7; 7; 2; 9]
             ce qui donne pour l : [2; 5; 4; 8; 2; 6; 7; 7; 9]
On effectue une passe sur la liste : [5; 4; 8; 2; 6; 7; 7; 9]
             ce qui donne pour l : [2; 2; 5; 4; 8; 6; 7; 7; 9]
On effectue une passe sur la liste : [5; 4; 8; 6; 7; 7; 9]
             ce qui donne pour l : [2; 2; 4; 5; 6; 8; 7; 7; 9]
On effectue une passe sur la liste : [5; 6; 8; 7; 7; 9]
             ce qui donne pour l : [2; 2; 4; 5; 6; 7; 8; 7; 9]
On effectue une passe sur la liste : [6; 7; 8; 7; 9]
             ce qui donne pour l : [2; 2; 4; 5; 6; 7; 7; 8; 9]
On effectue une passe sur la liste : [7; 7; 8; 9]
Ce qui donne  [2; 2; 4; 5; 6; 7; 7; 8; 9]
```

Vous remarquez que la dernière passe est inutile car il n'y a plus d'inversion, mais le programme doit s'en assurer. Une façon simple de s'arrêter à temps (pour éviter de boucler sur des passes inutiles) consiste en effet à adjoindre au résultat de la fonction `une_passe` un booléen qui signale si une transposition a été effectuée.

1° Programmez récursivement la fonction `une_passe`.

```
#une_passe [5; 4; 8; 2; 6; 7; 7; 2; 9];;
- : bool * int list = true, [2; 5; 4; 8; 2; 6; 7; 7; 9]
#une_passe [7; 7; 8; 9];;
- : bool * int list = false, [7; 7; 8; 9]
```

2° En déduire une implémentation récursive du tri bulle.

5. *Le tri fusion.*
Il s'agit à nouveau d'un tri suivant le paradigme diviser pour régner. Le principe en est le suivant :
– on divise en deux moitiés la liste à trier,
– on trie chacune d'entre elles,
– on fusionne les deux moitiés obtenues pour reconstituer la liste complète triée.

1° Écrivez une fonction `divise` récursive qui produit à partir d'une liste donnée deux moitiés de longueur égale (ou presque dans le cas d'une liste initiale de longueur impaire). Pour cela, inspirez-vous d'un croupier qui distribue un paquet de cartes à deux joueurs en remettant alternativement une carte à l'un puis à l'autre des deux adversaires.

```
#divise [5; 4; 8; 2; 6; 7; 7; 2; 9];;
- : int list * int list = [5; 8; 6; 7; 9], [4; 2; 7; 2]
```

2° Écrivez une fonction `fusion` qui fusionne deux listes triées en une seule.

```
#fusion [5; 6; 7; 8;9] [2; 2; 4; 7];;
- : int list = [2; 2; 4; 5; 6; 7; 7; 8; 9]
```

3° En déduire finalement la fonction `tri_fusion`.

6. Implémenter à présent, la recherche d'un chemin simple. Pour cela :

1° définir un type `point` contenant une étiquette `angle`,

2° initialiser une liste de points (utiliser le module `random`),

3° choisir un point d'ancrage (le premier de la liste) et calculer convenablement le champ `angle` de chacun des points,

4° adapter une des fonctions de tri ci-avant pour trier la liste suivant `angle`,

5° utiliser le module `graphics`, pour représenter le tout.

Vous verrez dans le troisième point, que le calcul brut de l'angle peut conduire à des erreurs... cela dit, l'angle exact n'est pas absolument nécessaire alors peut-être peut-on avantageusement le remplacer pour conserver le classement des points...