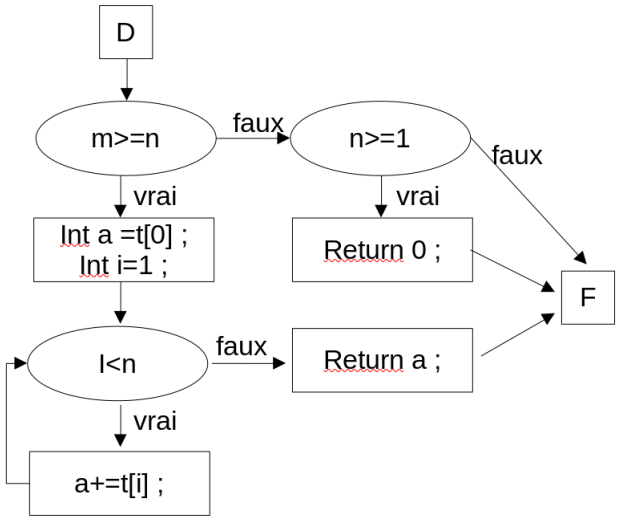
Devoir en temps limité n°4 – 3h

Calculatrices interdites

1 Questions de cours

1. *Qu'est-ce que la barrière d'abstraction ?*
   Programmer derrière la barrière d'abstraction, c'est programmer sans savoir comment une structure de données a été implémentée. On utilise que ses primitives.
   Cela permet la modularité du code : si quelqu'un fait une mise à jour du code de la structure, le programme derrière la barrière d'abstraction continue de fonctionner car il ne suppose que la correction des primitives.
2. *Quelles sont les primitives de la structure de données liste ?*
   - Créer une liste vide,
   - Tester si une liste est vide,
   - Récupérer la tête d'une liste,
   - Récupérer la queue d'une liste,
   - Ajouter un élément en tête d'une liste,
   - Supprimer un élément en tête d'une liste,
   - Désallouer toute la mémoire affectée à la liste.
3. *Qu'est-ce qu'un maillon dans une structure simplement chainée ?*
   Un maillon est une petite structure de données qui stocke une donnée et un pointeur vers le maillon suivant, ce qui permet de chainer la structure et odnc de la parcourir.
4. *Énoncer une implémentation possible d'une file et rappeler brièvement son fonctionnement, en utilisant un ou des dessins.*
   cf. cours
5. **Savoir-faire : Dessiner le graphe de flot de controle d'un code, Déterminer des jeux de tests qui couvrent les arcs**
   Dessiner le graphe de flot de contrôle du programme suivant et donner un jeu de tests permettant de couvrir tous les arcs. Quelle erreur peut on manquer, même avec un jeu de test qui couvre tous les arcs ?



   Le jeu de tests (m=-1, n=0, t un tableau vide), (m=1,n=2, t quelconque de taille 2), (m=12, n=3, t quelconque de taille 3) suffit pour tester tous les arcs.
   Ceci permet de repérer des erreurs : boucle while qui ne termine pas, on ne renvoie pas de valeur si $n < m$ et $n < 1$.
   Pour le test (m=42,n=0,t un tableau vide) l'instruction `int a = t[0];` est exécutée. Or elle provoque une erreur de segmentation. Notre jeu de tests précédent ne suffisait pas pour détecter cette erreur.

2 Exercices en C

Exercice 1 Implémentation d'une file circulaire en C

Savoir-faire : Implémenter une file à partir de maillons chainés et mener à bien l'implémentation quand on me propose des structures ou types adaptées, Déterminer la complexité d'un programme avec des boucles

On propose les types suivants :

```
typedef struct maillon {int valeur; maillon* suivant;} maillon;

typedef struct file {maillon* dernier;} file;
```

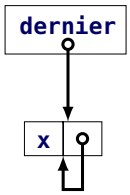
1. *Comment représenter la file vide avec cette implémentation ? Écrire la fonction* `file* creer()` *qui crée et renvoie un pointeur vers une file vide.*
   La file vide n'a pas de maillon. On représente donc la file vide par un pointeur NULL dans dernier.

```
file* creer(){
  file* f = malloc(siezof(file));
  f->dernier=NULL;
  return f;
}
```

2. Écrire la fonction `bool est_vide(file* f)` qui vérifie si la file est vide.

```
bool est_vide(file* f){
  assert(f!=NULL);
  return f->dernier == NULL;
}
```

3. *Faire un dessin de la file quand elle ne contient qu'un seul élément x. Écrire une fonction* `maillon* cree_maillon(int x)` *qui crée un maillon de valeur x qui pointe vers lui-même.*




```
maillon* cree_maillon(int x){
 maillon* res = malloc(sizeof(maillon));
 res->valeur = x;
 res->suivant = res;
 return res;
}
```

4. *Qu'a t-on oublié de vérifier dans la fonction* `defile` *proposée ? Proposer des commentaires pour la fonction.*
   On a oublié de vérifier que la file n'est pas vide. On peut le vérifier avec `assert(!est_vide(f));`

```
int defile(file* f){
 maillon* premier = f->dernier->suivant; //Le premier maillon est pointé par le dernier maillon
 int v = premier->valeur; //On récupere sa valeur, qu'on va renvoyer
 if(premier->suivant = premier){ //Si la liste est vide apres defilement, on met dernier a NULL
   f->dernier = NULL;
 }
 else{
   f->dernier->suivant = premier->suivant; /*On change la fleche du dernier pour qu'elle
 aille au nouveau premier*/
 }
 free(premier); //On libere la mémoire
 return v;
}
```

5. *Écrire la fonction* `int coupdoeil(file* f)` *qui regarde le prochain élément sans le défiler.*
   On déduit de defile.

```
int coupdoeil(file* f){
 assert(!est_vide(f));
 maillon* premier = f->dernier->suivant; //Le premier maillon est pointé par le dernier maillon
 int v = premier->valeur; //On récupere sa valeur, qu'on va renvoyer
 return v;
}
```

6. *Écrire la fonction* `void enfile(file* f, int x)`. *On fera attention au cas où la file est vide. On rappelle qu'on enfile à droite, donc l'élément enfilé devient le dernier élément.*

```
void enfile(file* f, int x){
 maillon* m = creer_maillon(x);
 if (est_vide(f)){
   f->dernier = m;
 }
 else{
   maillon* prec = f->dernier; //l'ancien dernier
   f->dernier = m; //m est le nouveau dernier
   m->suivant = prec->suivant; //m pointe vers le premier
   prec->suivant = m; //prec pointe vers m
 }
}
```

7. *Écrire la fonction* `int detruire(file* f)` *qui libère toute la mémoire affectée à la file.*

```
void enfile(file* f, int x){
 while(!est_vide(f)){
   defile(f); //defiler supprime un maillon
 }
 free(f); //on termine de libérer la mémoire
}
```

8. *Donner la complexité de chacune des fonctions précédentes.*
   À part dans `detruire` on a pas fait de boucle. Donc toutes les fonctions sont en temps constant sauf détruire qui est linéaire en la taille de la file au moment où on la détruit.

Exercice 2 Les palindromes **Savoir-faire : Utiliser une pile.**

1. *À l'étape d), que peut on dire si $c \neq c'$? Si $c = c'$?*
   Le principe de l'algorithme est que si $c' = s[i]$, alors c doit être $s[n-1-i]$. Si $c \neq c'$, la chaine n'est pas un palindrome. Si $c = c'$ par contre, on ne peut rien dire, il faut continuer l'algorithme pour savoir.
2. *Rappeler comment écrire une fonction* `int strlen(char* s)` *qui calcule la taille de la chaine* `s`.

```
int strlen(char* s){
  int i = 0;
  while (s[i]!='\0'){
    i+=1;
  }
  return i;
}
```

3. *Programmer la fonction* `bool est_palindrome(char* s)` *qui suit le principe expliqué précdemment.*

```
bool est_palindrome(char* s){
  pile* p = creer();
  int n = strlen(s);
  int i;
  for(i=0;i<n/2;i+=1){
    empiler(p, s[i]);
  }

  if(n%2==1){i+=1;}

  bool res = true;
  for(int j=i;j<n;j+=1){
    if(s[j]!=depile p){
      res = false;
    }
  }
  detruire(p);
  return res;
}
```

On va maintenant procéder à la preuve de ce programme.

Savoir faire : Écrire une spécification pour un programme, Prouver la terminaison, Trouver l'invariant sur des problèmes classiques, Prouver la correction totale d'un algorithme d'une fonction à une boucle, Déterminer la complexité d'un programme avec des boucles

4. *Écrire une spécification pour votre fonction.* Entrées : s une chaine de caractères.
   Sortie : true si s est un palindrome, false sinon.
5. Justifier que votre fonction termine. On a utilisé 2 boucles for, donc ça termine.

Un invariant pour la première boucle est "À la fin de l'étape i, la pile contient toutes les lettres de i à 0, dans l'ordre.". La preuve est évidente, on pourra donc supposer qu'au début de l'étape c) la pile contient toutes les lettres de $\lfloor \frac{n}{2} \rfloor - 1$ à 0, dans l'ordre.

6. *Proposer un (ou des) invariant(s) utile(s) pour la deuxième boucle. Conclure quant à la correction totale de la fonction.*
   Un invariant est "À la fin de l'itération j, **`res`** vaut **`true`** si et seulement si la chaine de caractères **`s[n-1-j]s[n-j] ... s[j]`** est un palindrome."
   Montrons cet invariant. À la fin de la première itération il est correct.
   - Supposons n pair, alors à la fin de la première boucle i valait $n/2$. Donc $j = n/2$.
     La condition de la boucle compare donc **`s[n/2]`** avec **`s[n/2-1]`** et **`res`** est vaut **`false`** si et seulement si **`s[n/2]`**$\neq$**`s[n/2-1]`**, donc si et seulement si **`s[n/2-1]s[n/2]`** n'est pas un palidrome.
   - Supposons n impair, alors à la fin de la première boucle i valait $\frac{n-1}{2}$ et il est augmenté de 1 par la condition entre les deux boucles. Donc $j = \frac{n+1}{2}$.
     La condition de la boucle compare donc **`s[(n+1)/2]`** avec **`s[(n-3)/2]`** et **`res`** est vaut **`false`** si et seulement si **`s[(n+1)/2]`**$\neq$**`s[(n-3)/2]`**, donc si et seulement si **`s[(n-3)/2]s[(n-1)/2]s[(n+1)/2]`** n'est pas un palidrome.

   Supposons l'invariant vérifié à la fin d'une itération j. On a deux cas possibles au début de l'itération $j+1$:
   - Soit **`res`** vaut déjà **`false`**, donc la chaine **`s[n-1-j]s[n-j] ... s[j]`** n'est pas un palindrome et il en est de même pour **`s[n-2-j]s[n-j-1] ... s[j+1]`**. **`res`** ne peut pas être remis à vrai.
   - Soit **`res`** vaut **`true`** et donc **`s[n-1-j]s[n-j] ... s[j]`** est un palindrome.
     On rappelle que puisque $j - i + 1$ éléments ont déjà été dépilés, **`depile p`** est le caractère en position $\lfloor \frac{n}{2} \rfloor - 1 - (j - i + 1) = \lfloor \frac{n}{2} \rfloor - 1 - (j - \lceil \frac{n}{2} \rceil + 1) = n - 1 - (j + 1)$. On raisonne maintenant selon la valeur du test **`s[j+1]!=depile p`** :
     - Si c'est vrai, alors **`s[n-1-(j+1)]=s[j+1]`**. Or **`s[n-1-j]s[n-j] ... s[j]`** est un palindrome, donc **`s[n-1-(j+1)]s[n-j-1] ... s[j+1]`** aussi. **`res`** reste par ailleurs vrai.
     - Sinon **`s[n-1-(j+1)]s[n-j-1] ... s[j+1]`** n'est pas un palidrome, et **`res`** est bien mis à faux.

   Notre propriété est bien invariante.
   Pour conclure, puisque la fonction termine, notre invariant est vrai à la fin, quand $j = n - 1$ et s'exprime "**`res`** vaut **`true`** si et seulement si la chaine de caractères **`s[0]s[n-j] ... s[n-1]`** est un palindrome.". Donc la fonction renvoie vrai si et seulement si la chaine est un palindrome.
7. *Quelle est la complexité de la méthode sachant que toutes les primitives s'effectuent en temps constant (sauf* `detruire` *qui est linéaire) ?*
   On note n la taille de s. strlen(s) est en $O(n)$.
   La première boucle fait un nombre d'itérations inférieur à n et à chaque itération, elle fait un nombre constant d'opérations élémentaires. Sa complexité est en $O(n)$.
   La deuxième boucle fait un nombre d'itérations inférieur à n et à chaque itération, elle fait un nombre constant d'opérations élémentaires. Sa complexité est en $O(n)$.
   La complexité totale est en $O(n)$.

3 Exercices en Ocaml

Exercice 3 Le tri à bulles

Savoir-faire : Comprendre un principe algorithmique nouveau. Prise d'initiative, Prouver un invariant donné

1. *Écrire une fonction* `parcours : 'a array -> bool` *qui prend en entrée un tableau t et fait* ***un parcours*** *du tableau en faisant les échanges nécessaires. La fonction renvoie un booléen indiquant si un échange a été fait lors du parcours.*

```
let parcours t =
  let n = Array.length t in
  let res = ref false in
  for i=0 to n-2 do
    if t.(i) > t.(i+1) then begin
      let sto = t.(i) in
      t.(i) <-t.(i+1);
      t.(i+1) <- sto;
      res := true
    end
  done; res;;
```

2. *Écrire une fonction* `tri_bulles : 'a array -> unit` *qui trie un tableau avec la méthode décrite.*

```
let tri_bulles t =
  let n = Array.length t in
  let res = ref true in
  while !res do
    res := parcours t
  done;;
```

3. *Montrer qu'un parcours n'effectue aucun échange si et seulement si t est trié.*
   Si t est trié, alors pour chaque i entre 0 et $n-2$, on a $t.(i) \geq t.(i+1)$ et donc on ne fait aucun échange.
   Supposons que le parcours de t ne fasse aucun échange, alors pour chaque i entre 0 et $n-2$ on a $t.(i) \geq t.(i+1)$, ce qui signifie que le tableau est trié.

Lemme (admis) : On note $a_0 \leq a_1 \leq \ldots \leq a_{n-1}$ les éléments d'un tableau t, ici numérotés selon leur valeur, pas selon leur position dans le tableau, qui peut être quelconque.

Si le sous tableau de t entre les indices $k \in [|0, n-1|]$ et $n-1$ est exactement $[|a_k; a_{k+1}; \ldots; a_{n-1}|]$, alors après un itération de `parcours` sur t, le sous tableau de t entre les indices $k-1$ et $n-1$ est exactement $[|a_{k-1}; a_k; a_{k+1}; \ldots; a_{n-1}|]$.

En particulier si le maximum a_{n-1} du tableau n'est pas en derniere position, alors le parcours met a_{n-1} dans la case $n-1$.

4. *Montrer la correction totale de votre algorithme.*
   Ici la méthode dévie de l'ordinaire. En effet, si on avait terminaison, alors d'après la question 3 on pourrait directement conclure que t est trié.
   Reste à montrer la terminaison et dans ce cas précis, il n'y a pas de variant évident. On va donc montrer un invariant qui nous aidera à trouver un variant.

   Soit t un tableau de longueur n et constitué des éléments $a_0 \leq a_1 \leq \ldots \leq a_{n-1}$ rangés dans un certain ordre. On montre l'invariant "À la fin du i-ème parcours, le sous-tableau entre les indices $n-i$ et $n-1$ (inclus) est trié et les éléments situés dans le reste du tableau sont inférieurs à $t[n-i]$".
   Au premier parcours, on sait que si le maximum de t n'était pas en dernière position, alors il l'est désormais. Donc le sous tableau entre les indices $n-1$ et $n-1$ est trié et tous les autres éléments du tableau sont inférieurs à $t[n-1] = a_{n-1}$ car c'est le maximum.

   Supposons maintenant avoir réalisé i parcours et que le sous-tableau entre les indices $n-i$ et $n-1$ (inclus) est trié et les éléments situés dans le reste du tableau sont inférieurs à $t[n-i]$.
   On effectue un $i+1$-ème parcours. Initialement le sous-tableau de t entre les indices $n-i$ et $n-1$ est exactement $[|a_{n-i}; \ldots; a_{n-1}|]$. Donc d'après le lemme après le $i+1$-ème parcours le sous-tableau de t entre les indices $n-i-1$ et $n-1$ est exactement $[|a_{n-i-1}; \ldots; a_{n-1}|]$.
   Ainsi le sous-tableau entre les indices $n-(i+1)$ et $n-1$ (inclus) est trié. De plus il n'existe que i éléments plus grands que a_{n-i-1} dans t, et ils sont tous à sa droite. On en déduit que tous les éléments situés dans le reste du tableau sont inférieurs à $t[n-(i+1)]$.

   Ainsi notre propriété est bien invariante. Le nombre d'éléments du tableau qui ne sont pas à leur place (celle qu'ils occupent dans la version triée du tableau) diminue strictement, puisqu'à chaque parcours l'indice séparant le tableau non-trié du tableau trié diminue d'au moins 1 d'après l'invariant.
   Remarque : Ici la condition de fin choisie est intimement liée au fait que le tableau soit trié. La seule chose qui nous permet de caractériser "à quel point le tableau est trié" est l'indice qui apparait dans l'invariant.

Exercice 4 Trier les copies

Savoir-faire : Utiliser une pile

On crée deux piles auxilaires : une pour les copies MP et une pour une copie CCINP. On vide la pile de copies en rangeant chaque copies dans la pile auxiliaire adaptée.

Finalement on vide la pile avec les copies ccinp dans la pile initiale, puis on vide la pile avec les copies mines-ponts. Remarque : on conserve ainsi l'ordre alphabétique.

```
let arrange p =
  let pmp = Stack.create() in
  let pccinp = Stack.create() in

  while not (Stack.is_empty p) do
    match Stack.pop p with
      |CCINP nom -> Stack.push (CCINP nom) pccinp
      |MP nom -> Stack.push (MP nom) pmp
  done;

  while (not Stack.is_empty pccinp) do
    Stack.push (Stack.pop pccinp) p
  done;

  while (not Stack.is_empty pmp) do
    Stack.push (Stack.pop pmp) p
  done;;
```

Exercice 6 Trier une pile

Utiliser une pile, Déterminer la complexité d'un programme avec des boucles

1. *Écrire une fonction* `insere : 'a -> 'a Stack.t -> unit` *qui prend en entrée un élément* **x** *et une pile* **p** *triée et insère l'élément* **x** *dans* **p** *de sorte à ce qu'elle reste triée. La pile doit contenir à la fin les éléments qu'elle contenait déjà, avec en plus* **x**.

   ```
   let insere x p =
     let paux = Stack.create () in
     while Stack.peek p > x do
       Stack.push (Stack.pop p) paux;
     done;

     Stack.push p x;

     while not (Stack.is_empty (Stack.peek paux)) do
       Stack.push (Stack.pop paux) p;
     done;;
   ```

2. *En déduire une fonction* `tri_pile : 'a Stack.t -> unit` *qui trie la pile.*

   ```
   let tri_pile x p =
     let paux = Stack.create () in
     while not (Stack.is_empty (Stack.peek paux)) do
       Stack.push (Stack.pop p) paux;
     done;

     while not (Stack.is_empty (Stack.peek paux)) do
       insere (Stack.pop paux) p
     done;;
   ```

3. *Quelle est la complexité du tri ? Préciser le pire cas et le meilleur cas.* On note n le nombre d'éléments dans la pile. La fonction insere vide un partie de p, ce qui coute au pire n opérations, puis rajoute un élément, c'est en $O(1)$. Enfin tous les éléments qui ont été sortis sont rerentrés, ce qui coute $O(n)$.
   La fonction tri_pile vide la pile, ce qui coute $O(n)$, puis utilise n fois insere, ce qui coute $O(n^2)$. La complexité est donc un $O(n^2)$ dans le pire cas.
   Le pire cas est si la pile est triée mais dans le mauvais sens originellement. Le meilleur cas est si la pile est triée dès le début. En effet insere a alors toujours une complexité de $O(1)$.

Exercice 7 Le tri de crêpes

Utiliser une pile, Comprendre une principe algorithmique nouveau, Déterminer la complexité d'un programme avec des boucles, Trouver l'invariant sur des problèmes classiques, Prouver la correction totale d'un algorithme d'une fonction à une boucle, Prise d'initiative

1. *Avant de commencer le tri il nous faut une fonction qui effectue le travail de la spatule. Écrire une fonction* `retourne : 'a Stack.t -> int -> unit` *qui prend en entrée un pile p et un entier i et retourne les i premières crêpes.*

```
let retourne p i =
  let sto = Array.make i (Stack.top p) in (*tableau accessoire pour stocker les éléments dépilés*)
  for j=0 to i-1 do (*On sort tous les éléments qui sont au-dessus de la spatule et on les met dans sto*)
    sto.(j)<-Stack.pop p
  done;
  for j=0 to i-1 do (*On remet les éléments au-dessus de la spatule sur la pile, a l'envers*)
    Stack.push sto.(j) p
  done;;
```

2. *Écrire une fonction* `indice_plus_grande_crepe : 'a Stack.t -> int` *qui cherche et renvoie quelle crepe est la plus grande. La crepe sur le dessus est numérotée 0, celle juste en dessous est numérotée 1, etc...*

```
let indice_plus_grande_crepe p =
  let sto = Stack.create in (*une pile auxiliaire*)
  let maxi = ref 0 in (*maximum*)
  let imaxi = ref 0 in (*indice de la crepe maximale*)
  let ind = ref 0 in (*indice de la crepe qu'on étudie*)
  while not (Stack.is_empty p) do (*On sort tous les éléments de la pile*)
    let el = Stack.pop p in
    if el>!maxi then begin (*et on les compare a maxi pour que maxi reste le plus grand élément*)
      maxi:=el;
      imaxi:=!ind
    end;
    ind:=!ind+1;
    Stack.push el sto
  done;

  while not (Stack.is_empty sto) do (*On remet p telle qu'elle était*)
    Stack.push (Stack.pop sto) p
  done; !imaxi;;
```

3. *Écrire une fonction* `tri_crepes : 'a Stack.t -> unit` *qui trie la pile de crêpes* ***en suivant le principe décrit***.
   Pour implémenter le principe décrit il nous faut la fonction `retourne` pour imiter le retournement, et il nous faut aussi une manière de déterminer la plus grande crêpe. La fonction implémentée précedemment ne suffit pas, puisqu'une fois la plus grande crêpe mis à sa place, il nous faut considérer la 2ème plus grande crêpe. (qui est la plus grande du "reste de la pile" comme dit l'algorithme)
   On écrit donc une variation de la fonction `indice_plus_grande_crepe` qui prend en entrée un indice n et ignore toutes les crêpes au delà de celle d'indice n.

```
let indice_plus_grande_crepe_bis p n =
  let sto = Stack.create () in (*une pile auxiliaire*)
  let maxi = ref 0 in (*maximum*)
  let imaxi = ref 0 in (*indice de la crepe maximale*)
  let ind = ref 0 in (*indice de la crepe qu'on étudie*)
  while (not (Stack.is_empty p)) && (!ind<=n) do (*On sort les éléments de la pile jusqu'a l'indice n*)
    let el = Stack.pop p in
    if el> !maxi then begin (*et on les compare a maxi pour que maxi reste le plus grand élément*)
      maxi:=el;
      imaxi:=!ind
    end;
    ind:=!ind+1;
    Stack.push el sto
  done;

  while not (Stack.is_empty sto) do (*On remet p telle qu'elle était*)
    Stack.push (Stack.pop sto) p
  done; !imaxi;;
```

   Finalement il nous faut aussi une fonction qui calcule la taille de la pile pour savoir comment retourner toute la pile (3eme étape). (en fait on peut le trouver avec `retourner` et `indice_plus_grande_crepe_bis` mais c'est un peu du bricolage)

```
let taille p =
    let sto = Stack.create () in (*une pile auxiliaire*)
    let res = ref 0 in (*la taille*)

    while not (Stack.is_empty p) do (*On dépile et on compte*)
      Stack.push (Stack.pop p) sto;
      res:=!res+1
    done;
```

```
    while not (Stack.is_empty sto) do (*On remet p telle qu'elle était*)
      Stack.push (Stack.pop sto) p
    done; !res;;
```

```
let tri_crepes p =
  let n = taille p in
  for i = 0 to n-1 do (*A chaque tour de boucle, i crepes ont déja été triées*)
    let j = indice_plus_grande_crepe_bis p (n-1-i) in (*On recherche la plus grande crepe non déja triée*)

    retourne p j; (*On retourne*)
    retourne p (n-1-i); (*On retourne encore, la plus grande crepe va en position n-1-i,
                          juste au-dessus de celles qui sont deja triées*)
  done;;
```

4. *Quelle est la complexité du tri ? Préciser le pire cas et le meilleur cas.*
   On note que la fonction `taille` est un $\Theta(t)$ où t est la taille de la pile et la fonction `indice_plus_grande_crepe_bis` est un $\Theta(n)$ où n est l'indice après lequel la fonction ignore les éléments (en supposant que celui-ci est plus petit que la taille de la pile). La fonction `retourne` quant à elle est un $\Theta(i)$ où i est son argument, la position de la spatule.
   À l'itération i, la boucle for dans `tri_crepes` effectue :
   - Une recherche de maximum parmi $n-1-i$ éléments, ce qui est linéaire en $t-1-i$.
   - Un retournement qui affecte j éléments. Dans le pire cas, j est toujours $t-1-i$, donc c'est linéaire en $n-1-i$.
   - Un retournement qui affecte $t-1-i$ éléments.

   Au total la complexité de chaque itération est majorée à partir d'un certain rang par $A_i * (t-1-i)$ avec A_i une certaine constante positive et minorée par $B_i * (t-1-i)$ avec B_i une certaine constante positive.
   On considère $A = max_i A_i$ et $B = min_i B_i$. Alors la complexité $C(t)$ de la boucle vérifie à partir d'un certain rang $B\sum_{i=0}^{t-1} t-1-i \leq C(t) \leq A\sum_{i=0}^{t-1} t-1-i$.
   Par un changement d'indice $j = t-1-i$ on se ramène à $B\sum_{j=0}^{t-1} j \leq C(t) \leq A\sum_{j=0}^{t-1} j$, ce qu'on peut simplifier :
   $B\frac{t^2-t}{2} \leq C(t) \leq A\frac{t^2-t}{2}$.
   Pour conclure proprement, on peut retirer le terme négatif à droite et indiquer que $t^2 - t \geq \frac{1}{2}t^2$ à partir du moment où $t \geq 2$.
   On a donc à partir d'un certain rang (plus grand que 2), $\frac{B}{4}t^2 \leq C(t) \leq \frac{A}{2}t^2$. On a montré que $C(t) = \Theta(t^2)$.
   Ici la complexité ne peut pas être mieux que $O(n^2)$, ni pire, puisque les appels `indice_plus_grande_crepe_bis p (n-1-i)` et `retourne p (n-1-i)` font le même nombre d'opérations peut importe la forme de la pile. La seule opération qui peut varier selon l'ordre des éléments dans la pile est `retourne p j`.
   Le meilleur cas est donc quand j est toujours le plus petit possible, c'est à dire quand la pile est dajà triée avec les plus grandes crêpes dans le haut. Le pire cas est quand j est toujours le plus grand possible, quand la pile est déjà triée.
5. *Faire la preuve de correction totale de l'algorithme de tri de crêpes.*
   Dans cette question on veut prouver l'algorithme lui-même, donc la fonction `tri_crepes`. On supposera que les fonctions `taille`, `indice_plus_grande_crepe_bis` et `retourne` sont correctes.
   La fonction `tri_crepes` termine puisqu'elle ne contient qu'une boucle for et des appels de fonctions qui terminent.
   Un invariant de la fonction est : "À la fin de l'itération i, les crêpes situées entre les indices $n-1-i$ et $n-1$ sont triées et les autres crêpes sont de diamètre plus petit que la crêpe en $n-1-i$".
   À l'itération $i = 0$, j vaut l'indice de la plus grande crêpe de la pile. `retourne p j` met la plus grande crêpe en haut de la pile et `retourne p (n-1-i)` la met en position $n-1-i = n-1$. Donc les crêpes entre les indices $n-1$ et $n-1$ sont triées (il n'y en a qu'une) et toutes les autres crêpes sont de taille plus petite puisque c'était la plus grande.
   Supposons le résultat à la fin d'une itération i et plaçons nous à l'itération $i+1$. j devient l'indice de la plus grande crêpe entre les indices 0 et $n-i-2$, qu'on appelera c.
   D'après l'invariant, le diamètre de cette crêpe est inférieur à celui de la crêpe en position $n-1-i$. Ensuite `retourne p j` met la crêpe c en haut de la pile et `retourne p (n-1-i)` la met en position $n-2-i$.

Les crêpes entre $n-1$ et $n-1-i$ n'ont pas été bougées par l'itération $i+1$. Puisqu'elles étaient triées elles le sont toujours, et puisque c est plus petite que chacune d'entre elles, la pile est maintenant triée entre les indices $n-1$ et $n-2-i$.

De plus par maximalité de c parmi les indices 0 à $n-2-i$, toutes les autres crêpes sont plus petites que celles déjà triées. Notre propriété tient toujours.

Pour conclure, l'invariant est toujours vrai à la fin de la dernière itération, pour $i = n-1$ et indique "Les crêpes situées entre les indices $n-1-(n-1)=0$ et $n-1$ sont triées et les autres crêpes sont de diamètre plus petit que la crêpe en $n-1-(n-1)$", ce qui est équivalent à "Toutes les crêpes sont triées.".

Ainsi notre fonction effectue bien un tri de la pile de crêpes.