Exercice 1 (quelques fonctions) :

1.
```c
int factorielle(int n)
{
  if (n < 0) { return -1; }
  int res = 1;
  for (int i = 1; i <= n; i++)
  {
    res *= i;
  }
  return res;
}
```

2.
```c
int suplog(int n)
{
  int d = 0;
  int p = 1; // p = 2^d
  while (n > p)
  {
    d++;
    p *= 2;
  }
  return d;
}
```

3.
```c
float somme(float *tab, int n)
{
  float s = 0.;
  for (int i = 0; i < n; i++)
  {
    s += tab[i];
  }
  return s;
}
```

4.
```c
void initialisation(int *tab, int n)
{
  for (int i = 0; i < n; i++)
  {
    tab[i] = i*i;
  }
}
```

5.
```c
bool chercher(int *tab, int x, int n)
{
  for (int i = 0; i < n; i++)
  {
    if (tab[i] == x)
    {
      return true;
    }
  }
  return false;
}
```

6.
```c
int *zeros(int n)
{
  int *tab = malloc(n * sizeof(int));
  for (int i = 0; i < n; i++)
  {
    tab[i] = 0;
  }
  return tab;
}
```

Exercice 2 (multiplication à la Russe) :

1. 69×135 :

Opération	Multiplicateur	Reste	Multiplicande	Résultat
	69	0	135	0
69/2	34	1	270	135
34/2	17	0	540	135
17/2	8	1	1080	675
8/2	4	0	2160	675
4/2	2	0	4320	675
2/2	1	0	8640	675
1/2	0	1	17280	9315

Donc $69 \times 135 = 9315$.

2.
```c
int mult_russe(int a, int b)
{
  int res = 0, mcateur = a, mcande = b;
  while (mcateur != 0)
  {
    if (mcateur%2 == 1)
    {
      res += mcande;
    }
    mcateur /=  2;
    mcande *= 2;
  }
  return res;
}
```

3. À chaque passage dans la boucle `while`, `mcateur` devient `mcateur / 2`, donc les valeurs successives de `mcateur` sont strictement décroissantes, et vont finir par atteindre 0 (et alors la boucle `while` termine). Donc la fonction `mult_russe` termine.

4. (a) Inv : $a \times b = \texttt{mcateur} \times \texttt{mcande} + \texttt{res}$

   (b) Montrons que Inv est un invariant de boucle.

   - Il est vrai avant la boucle, car $\texttt{mcateur} = a$; $\texttt{mcande} = b$; et $\texttt{res} = 0$.
   - Supposons que $a \times b = \texttt{mcateur} \times \texttt{mcande} + \texttt{res}$ en haut de la boucle, et montrons que $a \times b = \texttt{mcateur}' \times \texttt{mcande}' + \texttt{res}'$ en bas de la boucle (où v' désigne la nouvelle valeur d'une variable v).
     - si `mcateur` est pair, alors : $\texttt{mcateur}' = \frac{\texttt{mcateur}}{2}$, $\texttt{mcande}' = 2 \times \texttt{mcande}$, et $\texttt{res}' = \texttt{res}$. Donc $\texttt{mcateur}' \times \texttt{mcande}' + \texttt{res}' = \frac{\texttt{mcateur}}{2} \times 2 \times \texttt{mcande} + \texttt{res} = a \times b$ par HR.

- si `mcateur` est impair, alors : $\texttt{mcateur}' = \frac{\texttt{mcateur}-1}{2}$, $\texttt{mcande}' = 2 \times \texttt{mcande}$, et $\texttt{res}' = \texttt{res} + \texttt{mcande}$. Donc $\texttt{mcateur}' \times \texttt{mcande}' + \texttt{res}' = \frac{\texttt{mcateur}-1}{2} \times 2 \times \texttt{mcande} + \texttt{res} + \texttt{mcande} = \texttt{mcateur} \times \texttt{mcande} + \texttt{res} = a \times b$ par HR.

Donc Inv est un invariant de boucle.

De plus, en sortie de boucle `while`, on a $\texttt{mcateur} = 0$, donc $a \times b = 0 \times \texttt{mcande} + \texttt{res} = \texttt{res}$. L'algorithme est donc correct.

Exercice 3 (chaînes de caractères) :

1.
   - ligne 11 : `i <= n / 2;`
   - ligne 13 : `if (str[i] != str[n-1-i])`
2. Car `str` est déjà un pointeur vers la début d'une chaîne de caractères (il contient l'adresse du premier caractère de la chaîne).
3.
```c
int longueur(char *s)
{
  int i = 0;
  while (s[i] != '\0')
  {
    i++;
  }
  return i;
}
```

Exercice 4 (complexité) :

1. Dans la fonction `f1` :
   - la ligne 8 s'effectue en $O(1)$,
   - donc la boucle `for` de la ligne 6 s'effectue en $O(i)$,
   - donc la boucle `for` de la ligne 4 s'effectue en $O(n^2)$, car $\sum_{i=1}^{n/2} i = O(n^2)$.

   Donc `f1` a une complexité en $O(n^2)$.
2. Dans la fonction `f2` :
   - chaque passage dans la boucle `for` s'effectue en $O(1)$;
   - il reste à calculer combien de fois on va passer dans la boucle : au k-ème passage dans la boucle, $i = 2^k$, et on s'arrête quand $i \geq n$, i.e. $2^k \geq n$, donc $k \geq \log_2(n)$.

   Donc `f2` a une complexité en $O(\log_2(n))$.

Exercice 5 (pointeurs) :

1.

Variable	Adresse	Valeur
`x`	100	4
`p`	101	100
`p1`	102	103
`p2`	104	103
`y`	103	8
`*p`	—	4
`*p1`	—	8
`*p2`	—	8

2. En général, on utilise la fonction `malloc`, qui alloue un nouvel espace mémoire et renvoie l'adresse du début de cet espace.

Exercice 6 (piles) :

1. Ce code sert à vider le contenu de `p` dans `q`. Ainsi, après exécution de ce code, `p` est vide, et `q` contient les anciens éléments de `p` mais à l'envers :

2. Le code de la question 1 permet de parcourir tous les éléments de `p` tout en les sauvegardant dans `q`, mais il faut ensuite les remettre dans `p`.

   (a)
```c
int hauteur_pile(pile *p)
{
  pile* q = creer_pile();
  int n = 0;
  while (!pile_est_vide(p))
  {
    empiler(q, depiler(p));
    n++;
  }
  while (!pile_est_vide(q))
  {
    empiler(p, depiler(q));
  }
  return n;
}
```

   (b) La complexité de ce programme est en $O(n)$: chaque boucle `while` est exécutée une fois pour chaque élément de la pile de départ.

3. Dans le même esprit, on parcourt les i premiers éléments de `p` en les sauvegardant dans `q` pour chercher le max, puis on remet ces éléments dans `p`.

```c
int max_pile(pile *p, int i)
{
  pile* q = creer_pile();
  int j = 1;
  int x_max = sommet(p);
  for (int k = 1; k <= i; k++)
  {
    int x = depiler(p);
    empiler(q, x);
    if (x > x_max)
    {
      x_max = x;
      j = k;
    }
  }
  while (!pile_est_vide(q))
  {
    empiler(p, depiler(q));
  }
  return j;
}
```

4. L'astuce consiste à utiliser deux piles auxiliaires : on transfère d'abord les j premiers éléments de `p` dans `q1`, puis on vide `q1` dans `q2`, et enfin `q2` dans `p`.

```
void retourner(pile *p, int j)
{
  pile* q1 = creer_pile();
  for (int k = 1; k <= j; k++)
  {
    empiler(q1, depiler(p));
  }
  // on transfère les éléments de q1 dans q2, en les retournant
  pile* q2 = creer_pile();
  while (!pile_est_vide(q1))
  {
    empiler(q2, depiler(q1));
  }
  // on transfère les éléments de q2 dans p
  while (!pile_est_vide(q2))
  {
    empiler(p, depiler(q2));
  }
}
```

5. (a)

```
void tri_crepes(pile *p)
{
  int n = hauteur_pile(p);
  for (int i = n; i >= 1; i--)
  {
    int j = max_pile(p, i);
    retourner(p, j);
    retourner(p, i);
  }
}
```

(b) Tout comme pour `hauteur_pile`, les fonctions `max_pile` et `retourner` se font en $O(n)$. Ainsi, le contenu de la boucle `for` s'effectue en $O(n)$ (3 appels de fonctions de complexités linéaires), et on passe n fois dans cette boucle. D'où une complexité en $O(n^2)$.