h.0 (exercices_B, 1)

x	$x^2+1 \mod 11$
0	1
1	2
2	5
3	10
4	6
5	4
6	4
7	6
8	10
9	5
10	2

image :
1
2 (2x)
4 (2x)
5 (2x)
6 (2x)
10 (2x)

Raison : $\sqrt{n}$ a deux solutions.

Il n'est donc pas conseillé d'utiliser x^2 dans une fonction de hachage.

h.1 a) (exercices_A)

```
distincts_hachage(T):
    H = [2n]  # n = longueur de T
    for e in T:
    |   h = hash_fct(e)
    |   while H[h] != null and H[h] != e:
    |   |   h = h+1 mod 2n
    |   H[h] = e
    count = 0
    for x in H:
    |   if x != null:
    |   |   count ++
    return count
```

h.1 b) (exercices-A)

```
distincts_tri(T):
    H = build_heap(T) # taille n, temps O(n)
    count = 0
    last_min = null
    min = null
    while not H.is_empty():
        min = H.delete_min()
        if min != last_min:
            count ++
        last_min = min
    return count
```

h.2 Si on insère les valeurs dans l'ordre $x_1, x_2, \dots, x_{i+1}, x_i, \dots, x_n$ au lieu de $x_1, x_2, \dots, x_i, x_{i+1}, \dots x_n$, voici les cas possibles :

- x_i et x_{i+1} ne créent pas de collision. Dans ce cas, l'efficacité de recherche pour ces éléments reste la même (parfaite).

- x_i et/ou x_i+1 créent une/des collisions avec d'autres éléments. Comme on a seulement échangé l'ordre de deux éléments voisins, l'état final reste inchangé.

- x_i et x_{i+1} ont une collision ensemble : dans ce cas, une recherche infructueuse pour un élément qui a le même indice sera aussi efficace. Une recherche fructueuse pour x_{i+1} prendra 1 opération de moins et une recherche fructueuse pour x_i, 1 opération de plus. Donc globalement l'efficacité de recherche reste la même.

Comme toute permutation peut être produite par échange de h.2 (suite) voisins, le coût moyen de recherche reste le même peu importe l'ordre d'insertion des éléments. □

h.3 a) demarrage(T):

```
    Count = 0
    Somme = 0
    for i in 0..M-1 :  # M = longueur de T
        if T[i] != null:
            h = hash-fct(T[i])
            Count++
            Somme += i - h + 1 mod M
    return somme/count
```

b) classe Tableau_hachage:

```
    T = [M]  # capacité M
    n = 0
    c̄ = 0

    fct insert(x):
        n++
        h = hash_fct(x)
        c = 1
        While T[h] != x and T[h] != null and T[h] != FREE
            h = h + 1 mod M
            c ++
        T[h] = x
```

$$\bar{c} = \frac{\bar{c} \times (n-1) + c}{n}$$

h.3 b) (suite)

```
fct delete(x):
    n--
    h = hash_fct(x)
    c = 1
    while T[h] != x and T[h] != null:
    |   c++
    |   h = h+1 mod M
    if T[h] == null:
    |   n++
    |   Erreur("x not found!")
    T[h] = FREE
```

$$\bar{c} = \frac{\bar{c} \times (n+1) - c}{n}$$

h.4 a) Voir code Java.

b) Dans la boucle interne du tri fusion, le nombre d'échanges à faire correspond au nombre d'opérations pour la recherche fructueuse de l'élément en question. Avec $\alpha = 1/2$, ce nombre est $\sim \frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right) = \sim 1.5$. Donc l'étape de tri prend $\sim 1.5n$ opérations, $\in O(n)$. $\square$

exercices_B, 2

k	$h(k) = \lfloor M \cdot \{\frac{\sqrt{5}-1}{2} k\} \rfloor$
10	2
22	7
31	1
4	5
15	3
28	3
17	6
88	4
65	2

	0	1	2	3	4	5	6	7	8	9	10	11
a)		31	10	15	28	4	17	22	88	65		
b)		31	10	15	28	4	17	22	88			65

exercices_C

Cette procédure ne cause pas de grappe forte car elle distribue les éléments qui ont le même code de hachage à de grandes distances les uns des autres.

Par contre, comme la taille est de 2^{20} et que les sauts sont des multiples de $1024 = 2^{10}$, il y a seulement 2^{10} cases possibles pour assigner un même code de hachage. Cela pourrait poser problème s'il y a un nombre élevé de collisions pour le même code. Solution : prendre M = grand nombre premier loin d'une puissance de 2.

exercices _D a) ADT : Multiset ou "sac". Politique d'adressage : sondage linéaire.

b) fct delete(x):

```
i = h(x)
while H[i] != x :
|  x = x + 1 mod M
H[i] = DELETED
```

fct insert(x):

```
i = h(x)
while H[i] != null and H[i] != DELETED :
|  i = i + 1 mod M
H[i] = x
```

fct search(x): rien à modifier.

c) fct delete(x):

```
i = h(x)
while H[i] != x and H[i] != null :
|  i = i + 1 mod M
if H[i] = x :
|  H[i] = null
|  i = i + 1 mod M
|  while H[i] != null:
|  |  y = H[i]
|  |  H[i] = null
|  |  insert(y)
|  |  i = i + 1 mod M
```