I. Généralités

Des volumes importants de données sont susceptibles d'être traitées par les ordinateurs. Des algorithmes efficaces sont alors nécessaires pour réaliser ces opérations comme, par exemple, la sélection et la récupération des données. Les algorithmes de recherche entrent dans cette catégorie. Leur rôle est de déterminer si une donnée est présente et, le cas échéant, d'en indiquer sa position, pour effectuer des traitements annexes. La recherche d'une information dans un annuaire illustre cette idée. On cherche si telle personne est présente dans l'annuaire afin d'en déterminer l'adresse. Plus généralement, c'est l'un des mécanismes principaux des bases de données : à l'aide d'un identifiant, on souhaite retrouver les informations correspondantes.
Dans cette famille d'algorithmes, la recherche dichotomique permet de traiter efficacement des données représentées dans un tableau de façon ordonnée.

II. Présentation de l'algorithme

1) Approche naïve

Une première idée est de parcourir l'ensemble du tableau :

```python
def recherche_naive(tab, val):
    """Renvoie un indice correspondant à une position de la valeur val dans le tableau tab.
    Si la valeur n'est pas présente, renvoie None."""
    for .................... :
        if .............. :
            ..........
    ...........
```

Si la liste comporte n éléments, au pire il va y avoir ... tours de boucle. La complexité est donc ....

2) Approche par dichotomie

Le principe est celui qu'on a tous appliqué étant enfant pour deviner un nombre en ayant comme réponse à nos propositions « c'est plus petit » ou « c'est plus grand » : on compare la valeur cherchée à celle située ....
Si elle est .... , on peut restreindre à ....
Si elle est .... , on peut restreindre à ....
En répétant ce procédé, on divise la zone de recherche par .... à chaque étape. On dit que l'on procède par dichotomie, du grec *dikha*(en deux) et *tomos*(couper). C'est le principe informatique connu sous le nom *diviser pour régner* sur lequel on reviendra en terminale.
Après un certain nombre d'étapes, on finira soit par .... , soit par ....

<u>**Attention**</u> : Cette méthode ne marche que si le tableau dans lequel on cherche est ....

```python
def recherche_dichotomique(tab, val):
    """Renvoie un indice correspondant à une position de la valeur val dans le tableau trié tab.
    Si la valeur n'est pas présente, renvoie None."""
    g = .
    d = .............
    while ...... :
        m = .............
        if ............. :
            ..........
        elif ............. :
            ..........
        else:
            ..........
    return .....
```

Exemple : On recherche la valeur 42 dans le tableau suivant :

valeurs	13	25	38	43	43	52	53	64	74	87	89	91	93
indices	0	1	2	3	4	5	6	7	8	9	10	11	12

Initialisation : `g = ..` ; `d = ....`
1^e^ tour de boucle : `m = ..` ; `tab[m] = ......` 42 donc on continue avec `g = ...` et `d = ...`
2^e^ tour de boucle : `m = ..` ; `tab[m] = ......` 42 donc on continue avec `g = ...` et `d = ...`
3^e^ tour de boucle : `m = ..` ; `tab[m] = ......` 42 donc on continue avec `g = ...` et `d = ...`
4^e^ tour de boucle : `m = ..` ; `tab[m] = ......` 42 donc on continue avec `g = ...` et `d = ...`
Comme ...., on sort de la boucle et on renvoie ....

III. Analyse de l'algorithme

1) Terminaison

Pour prouver la terminaison de l'algorithme, on cherche un ....
Ici, on peut prendre .... En effet c'est :

- ....
- ....
- ....
  ....
  ....
  ....

2) Correction

Tout d'abord, on peut remarquer qu'il n'y a que deux façons de quitter la fonction : par le `return` de la ligne 13 ou celui de la ligne 14. Dans le premier cas, on a `tab[m]=` ... et la valeur renvoyée est donc correcte. Dans le second cas, on doit prouver que `val` n'est pas présente dans le tableau. Pour cela, on cherche un ....
Ici, on va utiliser la propriété : « Si `val` est présente dans `tab`, c'est .... ».
On peut se représenter la situation par ce dessin :

	0	g	m	d	len(tab)-1
t	éléments < `val`	···	?	···	éléments > `val`

Initialisation : Avant le premier tout de boucle, `g = ..` et `d = .................`. Comme tous les indices des éléments du tableau sont entre ces deux valeurs, la propriété est vérifiée.

Hérédité : Supposons qu'au début d'un tour de boucle, la propriété soit vérifiée. Si `tab[m] == val`, la fonction va renvoyer .. qui est bien un indice valable comme on l'a déjà vu. Sinon, on distingue deux cas :

- Si `tab[m] < val`. Comme le tableau est trié, on a pour tout $0 \leqslant i \leqslant m$: ....
  En notant `g' = ..........` et `d' = ...` les valeurs de `g` et`d` à la fin du tour de boucle, on en déduit que si `val` est présente dans `tab`, c'est ....

	0 m	g' d'	len(tab)-1
t	éléments < `val`	···	éléments > `val`

- Si `tab[m] > val`. Comme le tableau est trié, on a pour tout $m \leqslant i \leqslant$ `len(t)-1` : ....
  En notant `g' = ..` et `d' = ......` les valeurs de `g` et`d` à la fin du tour de boucle, on en déduit que si `val` est présente dans `tab`, c'est ....

	0	g' d' m	len(tab)-1
t	éléments < `val`	···	éléments > `val`

Conclusion À la fin du dernier tour de boucle, si `val` est présente dans `tab`, c'est ....
Or on a `g` ... `d` donc il n'y a aucun élément entre les indices `g` et `d`, ce qui prouve que `val` n'est pas dans `tab`.

3) Complexité

<u>**Exemple**</u> : Supposons qu'on cherche une valeur dans un tableau de taille 100. Plaçons nous dans le pire des cas, c'est à dire lorsque .... À la fin de la première itération, on va restreindre la recherche à un tableau de taille .... En restant dans le pire cas, à la seconde itération on aura .... éléments, à la troisième ... , *etc.*
Au total, il faudra au pire ... itérations, ce qui signifie qu'on n'examinera pas plus de ... valeurs dans le tableau.

Plus généralement, supposons qu'il existe un entier k tel que la taille du tableau soit inférieure ou égale à 2^k. À la fin de la première itération, on restreint à ...... éléments, à la deuxième itérations ...... éléments, *etc.* Au total, il faudra donc au maximum ... itérations. On a vu que cette complexité est .... , ce qui est très efficace. Par exemple pour un tableau d'un milliard d'éléments, il suffit de .... itérations.