1 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 (ex : recherche si telle personne est présente dans un annuaire afin d'en déterminer l'adresse).
Dans cette famille d'algorithmes, la **recherche dichotomique** permet de traiter efficacement des données représentées dans un tableau de façon ordonnée.

2 Présentation de l'algorithme

2.1 Approche séquentielle

Nous avons déjà vu une première façon de rechercher une valeur dans un tableau à l'aide d'un parcours séquentiel de tableau :

```python
def recherche_sequentielle(tab, val):
  for i in range(len(tab)):
    if tab[i] == val:
      return i
  return -1
```

Ici, on renvoie un entier (≥ 0) qui correspond à la position de la valeur recherchée dans la tableau en cas de succès, et -1 en cas d'échec. Comme tout algorithme ayant cette forme, la **complexité est linéaire**.
Mais **si on travaille avec un <u>tableau ordonné</u>**, il est possible d'être beaucoup plus efficace ! On retrouve ici un exemple de l'intérêt des algorithmes de tri.

2.2 Approche par dichotomie

L'idée centrale de cette approche repose sur l'idée de **réduire de moitié l'espace de recherche à chaque étape** : on regarde la valeur du milieu et si ce n'est pas celle recherchée, on sait qu'il faut continuer de chercher dans la première moitié ou dans la seconde.
On procède ainsi :

1. on déterminer l'élément m au milieu du tableau ;
2. si c'est la valeur recherchée, on s'arrête avec un succés ;
3. sinon, deux cas sont possibles :
   (a) si m est plus grand que la valeur recherchée, il suffit de continuer à chercher dans la première moitié du tableau ;
   (b) sinon, il suffit de chercher dans la deuxième moitié.
4. on répète cela jusque avoir trouvé la valeur recherchée, ou bien avoir réduit l'intervalle de recherche à un intervalle vide, ce qui signifie que la valeur recherchée n'est pas présente.

À chaque étape, on coupe l'intervalle de recherche en deux, et on en choisit une moitié. On dit que l'on procède par dichotomie, du grec dikha (en deux) et tomos (couper).

L'exemple très connu illustrant cette méthode est le jeu « devine un nombre entre 1 et 100 ».

2.3 Une implémentation en Python

```
def recherche_dichotomique(tab, val):
  """ Renvoie la position de val dans tab  (ou -1).
  préconditions : - tab est un tableau trié
                  - val : valeur recherchée dans le tableau
  """
  # les variables gauche, droite et milieu correspondent à des indices
  gauche = 0
  droite = len(tab) - 1
  while droite - gauche >= 0:
    milieu = (gauche + droite) // 2
    if tab[milieu] == val:
      # on a trouvé val dans le tableau,  à la position milieu
      return milieu
    elif tab[milieu] > val:
      # on cherche entre gauche et (milieu - 1)
      droite = milieu - 1
    else: # on a tab[milieu] < val
      # on cherche entre (milieu + 1) et droite
      gauche = milieu + 1
  # on est sorti de la boucle sans trouver val
  return -1
```

3 Analyse de l'algorithme

Pour s'assurer que le programme ci-dessus fonctionne correctement, il faut se poser deux questions importantes :

1. Le programme renvoie-t-il bien un résultat ? Comportant une **boucle non bornée**, est-on sûr d'en sortir à un moment donné ?
2. La réponse renvoyée par le programme est-elle correcte ?

Remarque : l'intégralité de l'analyse de cet algorithme n'est pas au programme.

3.1 Terminaison du programme (à connaître)

La fonction `recherche_dichotomique` contient une boucle non bornée, une boucle while, et pour être sûr de toujours obtenir un résultat, il faut s'assurer que le programme termine, que l'on ne reste pas bloqué infiniment dans la boucle.
Pour prouver que c'est bien le cas, nous allons utiliser un **variant de boucle** (voir cours annexe).

Le variant de boucle est (`droite - gauche`).

Montrons que le variant décroît strictement lors de l'exécution du corps de la boucle.
On commence par définir `milieu = (gauche + droite) // 2`.
Deux cas sont alors possibles :

1. Cas 1 : `tab[milieu] == val`, on sort directement de la boucle à l'aide d'un `return` : la terminaison est assurée.
2. *Cas 2a* : `tab[milieu] > val`, on modifie la valeur de `droite` en la diminuant à (`milieu-1`) ce qui réduit l'intervalle (`droite - gauche`)
   Cas 2b : `tab[milieu] < val`, on modifie la valeur de `gauche` en l'augmentant à (`milieu+1`) ce qui réduit l'intervalle (`droite - gauche`)
   Ainsi, le variant a strictement décru pour les cas 2a et 2b.

Ayant réussi à exhiber un variant pour notre boucle, nous avons prouvé qu'elle termine bien.

3.2 Complexité (à titre informatif, hors programme)

Pour un tableau de taille $\boldsymbol{N}$, **la complexité de cet algorithme est en** $\mathbf{log_2(\boldsymbol{N})}$, càd un **coût logarithmique** : Cela signifie que si on note $\boldsymbol{n}$ le nombre d'itérations à effectuer, on aura dans le pire des cas $\mathbf{2^{\boldsymbol{n}} = \boldsymbol{N'}}$ (N' est la puissance de 2 la plus proche supérieure à N).
(Remarque : la fonction logarithme (log) sera étudiée en cours de math pour les spécialistes)

Démonstration :
On note n le plus petit entier tel que $2^n \geq N$.
(exemple : si $N = 27$, $n = 5$ car $16 < 27 \leq 32$ et $32 = 2^5$).
Puisque qu'à chaque itération, on sélectionne une moitié de ce qui reste, après la 1ère itération, il reste au maximum $2^n/2 = 2^{n-1}$ valeurs à tester. Après 2 itérations, il en reste au maximum $2^n/4 = 2^{n-2}$, et au bout de n itérations, il reste au maximum $2^n/2^n = 2^{n-n} = 1$ valeur à tester, et c'est donc terminer !

On constate que n est donc bien le nombre maximum d'itérations que l'on doive effectuer dans cet algorithme. (remarque : avec de la chance, on trouve la valeur recherchée plus rapidement).

Remarque : $\log_2(N)$ (plus précisément sa partie entière supérieure) donne aussi le nombre de bits de N écrit en binaire.

3.3 Correction du programme (facultatif, suggéré par le programme)

Deux cas sont à considérer pour prouver la correction, suivant que la valeur recherchée se trouve ou non dans le tableau.

Dans le cas où l'algorithme est arrêté prématurément par le `return` (dans la boucle while), il est évident que le résultat renvoyé est correct puisque l'exécution de ce `return` est subordonné au test `tab[milieu] == val`.

Considérons maintenant le cas où le programme renvoie -1, indiquant ainsi que la valeur recherchée n'est pas présente dans le tableau.

Pour prouver cela, nous allons utiliser un **invariant de boucle**.
Rappel : un invariant de boucle est une propriété P qui a le comportement suivant :
Si P est vérifiée au début de l'exécution du corps de la boucle, alors P est encore vérifiée à la fin de l'exécution du corps de la boucle.

L'exécution d'une boucle while s'effectue de la façon suivante : on commence par effectuer le test, et si le résultat du test est True, on effectue le corps de la boucle et on recommence (en retournant au test).
En conséquence, si P (invariant de boucle) est vérifiée en entrant dans la boucle lors du premier test, alors tant que le test donne une réponse positive, la préservation de P lors de l'exécution du corps de la boucle fait que la propriété sera toujours vérifiée à chaque nouveau test.
Et lorsque la condition du test n'est plus vérifiée et que l'on sort de la boucle, la propriété P sera encore vérifiée à ce moment-là.
En résumé, si une propriété P est vérifiée en entrant dans une boucle while, et si c'est un invariant de la boucle, alors P est encore vérifiée en sortant de la boucle.

Retour à la preuve de la correction lorsque la valeur n'est pas présente :
Pour prouver que si le programme renvoie -1 , alors c'est bien que la valeur recherchée n'est pas présente dans le tableau, nous allons utiliser l'invariant de boucle suivant :
Inv : « Si `val` est présente dans `tab` , c'est nécessairement à un indice compris entre `gauche` et `droite` (inclus) ».

Prouvons qu'il s'agit bien d'un invariant de la boucle de la fonction `recherche_dichotomique`.
Pour cela, il faut s'intéresser à une exécution qui va jusqu'à la fin du corps de la boucle : on ne tient pas en compte de cas où l'on sort prématurément à l'aide d'un return. On suppose donc qu'en entrée de boucle, **Inv** est vérifié. Après avoir défini milieu, trois cas sont examinés :

1. si `tab[milieu] == val` , on sort de la boucle prématurément à l'aide de l'instruction return, donc ce cas ne nous intéresse pas dans le cadre de la preuve d'invariant de boucle.
2. si `tab[milieu] > val`, et si `val` est présente dans le tableau, alors comme celui-ci est ordonné de façon croissante, cela implique que `val` ne peut être présent à l'indice `milieu`, ni après. On en déduit d'après **Inv** que si `val` se trouve dans `tab` c'est nécessairement à un indice compris entre `gauche` et `(milieu - 1)`. Ainsi, après l'affectation `droite = (milieu - 1)`, **Inv** est encore vérifié.
3. sinon, on a `tab[milieu] < val` et, de même, cela implique que l'on ne peut trouver `val` dans le tableau à un indice inférieur ou égal à `milieu`. Ainsi, en supposant **Inv** vrai au départ et en effectuant l'affectation `gauche = (milieu + 1)`, **Inv** reste vérifié.

Ainsi, dans tous les cas d'une exécution menant à la fin du corps de la boucle, si **Inv** est vérifié au début du corps, il l'est encore à la fin. C'est donc un invariant de la boucle.

Voyons maintenant comme cela nous permet de prouver la correction de cette fonction lorsque le résultat est -1.
Avant d'entrer dans la boucle, on a `gauche = 0` et `droite = (len(tab) - 1)`, et donc si `val` est présente dans `tab`, c'est nécessairement à un indice compris entre `gauche` et `droite`. Ainsi, **Inv** est vérifié en entrant dans la boucle.
Et puisqu'il s'agit d'un invariant de cette boucle, il est encore vérifié en sortant de la boucle. Mais à ce moment, `(droite - gauche) >= 0` n'est plus vérifiée (échec du test du while), on a

donc `droite < gauche`. Or d'après **Inv**, si `val` est présente dans `tab` à une position `pos`, alors on a `gauche <= pos <= droite` ce qui est incompatible avec `droite < gauche`.
Ainsi, en sortie de boucle, `val` ne peut être présente dans le tableau (car aucun indice n'est possible) et le résultat renvoyé -1 est donc correct.

En conclusion, la fonction `recherche_dichotomique` a deux comportements possibles : si le résultat renvoyé est un entier positif, on a montré qu'il correspond à un indice où se trouve la valeur recherchée dans le tableau ; sinon, le résultat renvoyé est -1 et on a montré que dans ce cas, la valeur recherchée n'apparaît pas dans le tableau (ouf !)