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 la recherche de données dans une table.
Dans cette famille d'algorithmes, la **recherche dichotomique** permet de traiter efficacement des données représentées dans un tableau trié.

2 Présentation de l'algorithme

2.1 Approche séquentielle

Rappel : Nous avons déjà vu une première façon de rechercher une valeur dans un tableau à l'aide d'un parcours séquentiel de tableau (chap. P7-1). Comme tout algorithme ayant cette forme, la **complexité est linéaire**.
Mais **si on travaille avec un tableau trié**, 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 l'approche par **dichotomie** repose sur le fait 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.

La structure de l'algorithme est la suivante :

1. on détermine l'élément m au milieu du tableau ;
2. si c'est la valeur recherchée, on s'arrête avec un succès ;
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 jusqu'à 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 ».

```
- 50 ?
- moins.
- 25 ?
- plus.
- 37 ?
- plus.
- 43 ?
- moins.
- 40 ?
- plus.
- 41 ?
- plus => 42 !
```

2.3 Une implantation en Python

```python
def recherche_dichotomique(tab, val):
  """ Renvoie la position de val dans tab (ou -1 si échec de recherche).
  précondition : tab est un tableau trié.
  """
  igauche = 0                        # borne de recherche inférieure
  idroite = len(tab) - 1             # borne de recherche supérieure
  while igauche <= idroite:
    imilieu = (igauche + idroite) // 2
    if tab[imilieu] == val :         # val trouvée dans le tableau
      return imilieu
    elif tab[imilieu] > val:         # on doit chercher entre igauche et (imilieu - 1)
      idroite = imilieu - 1
    else:                            # on doit chercher entre (imilieu + 1) et idroite
      igauche = imilieu + 1
  return -1                          # sortie de boucle sans trouver val
```

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, et pour s'assurer que le programme termine, nous devons exhiber un variant de boucle.

Le variant de boucle est `(idroite - igauche)`.

Vérifions que ce variant décroît strictement lors de l'exécution du corps de la boucle.
Deux cas sont alors possibles :

1. Cas 1 : `tab[imilieu] == val`, on sort prématurément de la boucle à l'aide d'un `return` : la terminaison est assurée.
2. Cas 2a : `tab[imilieu] > val`, on modifie la valeur de `idroite` en la diminuant à `(imilieu-1)` ce qui réduit l'intervalle `(idroite - igauche)`.
   Cas 2b : `tab[imilieu] < val`, on modifie la valeur de `igauche` en l'augmentant à `(imilieu+1)` ce qui réduit l'intervalle `(idroite - igauche)`.
   Ainsi, le variant décroît bien pour les cas 2a et 2b.

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

3.2 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[imilieu] == 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**.

Invariant : « Si `val` est présente dans `tab` , c'est nécessairement à un indice compris entre `igauche` et `idroite` (inclus) ».

Prouvons qu'il s'agit bien d'un invariant de la boucle while de la fonction `recherche_dichotomique`.
On suppose donc qu'en entrée de boucle, **Invariant** est vérifié. Après avoir défini `imilieu`, trois cas sont examinés :

1. si `tab[imilieu] == 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 `val < tab[imilieu]`, et si `val` est présente dans le tableau, alors comme celui-ci est trié, cela implique que c'est nécessairement à un indice compris entre `igauche` et `(imilieu - 1)`. Ainsi, après l'affectation `idroite = (imilieu - 1)`, **Invariant** reste encore vérifié.
3. enfin, si `tab[imilieu] < val`, et si `val` est présente dans le tableau, cela implique que c'est nécessairement à un indice compris entre `(imilieu + 1)` et `idroite`. Ainsi, après l'affectation `igauche = (imilieu + 1)`, **Invariant** reste encore vérifié.

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

Voyons maintenant comment cela nous permet de prouver la correction de cette fonction lorsque le résultat est -1.
Lorsque la boucle while se termine, c'est parce que `igauche > idroite`. Or **Invariant** nous dit que si `val` est dans le tableau, c'est entre `igauche` et `idroite`. Mais puisque cet intervalle est devenu vide, c'est que `val` n'est pas présente dans le tableau !

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.

3.3 Complexité (à titre informatif, hors programme)

Pour un tableau de taille N, **la complexité de cet algorithme est en** $\log_2(N)$, on dit que le **coût est logarithmique**.

Cela signifie qu'en doublant la taille des entrées (ici, la taille du tableau), il ne faut qu'une itération de plus pour résoudre le problème.

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