Recherche dichotomique

On va encore chercher longtemps ?

Lorsqu'on veut savoir si une valeur `val` se trouve dans un tableau `tab`, la méthode naïve consiste à parcourir ce tableau jusqu'à trouver la valeur. Si à la fin du parcours on ne l'a pas trouvée, on peut renvoyer une valeur d'erreur, comme `None`.

```
def recherche_naive(val, tab):
    for i in range(len(tab)):
        if val == tab[i]:
            return i
    return None
```

Si la valeur cherchée est au début du tableau, la recherche ira très vite. Mais dans le pire des cas, il faut parcourir tout le tableau. Le coût est donc linéaire. Le temps de calcul évolue proportionnellement avec la taille du tableau. Si on doit chercher dans un tableau quelconque, on ne peut pas faire mieux. Mais si le tableau est trié, il est possible d'aller bien plus vite. Mais avant de voir cet algorithme, il faut jouer un peu.

Je pense à un nombre...

Prenons l'exemple d'un jeu pour expliquer comment chercher une valeur de façon astucieuse. Imaginons qu'un joueur pense à un nombre entre 1 et 63 et qu'un deuxième essaie de le deviner. Ce dernier propose des nombres et l'autre lui répond en lui disant si le nombre secret est plus grand ou plus petit. L'objectif est de deviner le nombre en faisant le moins d'essais possibles. Bien entendu, si on a de la chance, on peut trouver le nombre en un seul coup. Mais dans le pire des cas, combien de tentatives faut-il ?

Exercice 1 : On doit deviner un nombre entre 1 et 63.

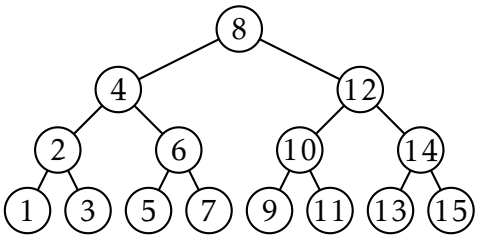
1) Si on donne les nombres dans l'ordre, un par un : 1, 2, 3, ... Combien d'essais faut-il dans le pire des cas ?
2) Au premier essai, on propose 50.
   a) Si on nous dit que le nombre secret est plus grand, quelles sont les valeurs encore possibles ?
   b) Si au contraire le nombre est plus petit, quelles sont les valeurs encore possibles ?
   c) Lequel des deux cas est le pire ?
3) Expliquer pourquoi 32 est la meilleure valeur à proposer en premier ?
4) Si le nombre cherché est plus petit, quelle doit être la deuxième proposition ?
5) Quelle est la stratégie à adopter pour minimiser le nombre d'essais nécessaires dans le pire des cas ?
6) Quel est le nombre maximum de propositions nécessaires dans le pire des cas ?

Cette stratégie s'appelle la **dichotomie**. Elle consiste à diviser en deux l'espace de recherche à chaque étape jusqu'à trouver le résultat recherché. Le fait de diviser va beaucoup plus vite que d'enlever les valeurs une à une, ce qui explique l'efficacité de cette approche. L'algorithme qui permet de gagner à ce jeu se trouve ci-contre.

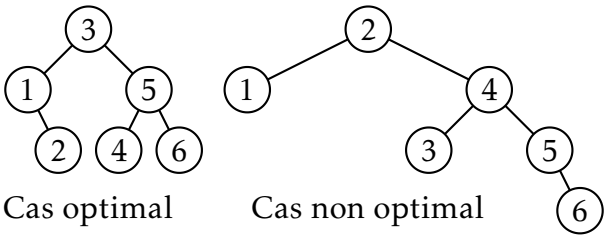
$g \leftarrow 1$
$d \leftarrow 63$
Tant que pas Trouvé **:**
  $m \leftarrow \left\lfloor \frac{g+d}{2} \right\rfloor$
  Proposer m
  Si Trouvé **:**
    Bravo !
  Si C'est moins **:**
    $d \leftarrow m-1$
  Si C'est plus **:**
    $g \leftarrow m+1$

Tout est une question d'équilibre

La stratégie par dichotomie est optimale pour ce jeu. Elle permet de minimiser le nombre de coups nécessaires dans le pire des cas. On peut représenter les propositions possibles à l'aide d'un arbre, comme celui ci-contre pour les nombres de 1 à 15. On part du haut et si le nombre cherché est plus petit, on prend la branche de gauche. Sinon, on prend la branche de droite, et on répète jusqu'à trouver le nombre.



On remarque que dans le pire des cas, il faut faire 4 propositions. On obtient un arbre parfaitement équilibré. Dans l'exemple ci-contre, on voit qu'une stratégie non optimale déséquilibre l'arbre et augmente le pire des cas.



Alors, comment déterminer le nombre de propositions nécessaires dans le pire des cas ? On remarque qu'avec une seule proposition, on ne peut trouver qu'un seul nombre. Avec 2, on peut trouver le bon parmi 3. Le nombre de nombres que peut contenir chaque étage est une puissance de 2. Ainsi, à l'étage k, il y a 2^{k-1} nombres. Un arbre de k étages peut donc contenir $1+2+4+\ldots+2^{k-1} = 2^k - 1$ nombres.
Pour trouver la bonne réponse parmi n nombres, il faut k propositions, où k est le plus petit entier tel que $n < 2^k$. Le nombre de propositions est donc le logarithme de n, arrondi par excès. Le coût de cet algorithme est logarithmique. C'est pour cela qu'il est très efficace.

Recherche dichotomique dans un tableau trié

Lorsqu'on a un tableau trié dans l'ordre croissant, il est possible d'appliquer une **recherche dichotomique**. Le principe est de choisir l'indice au milieu du tableau, de regarder si la valeur est plus petite ou plus grande que la valeur cherchée, et de recommencer dans la moitié du tableau correspondante. On obtient alors une recherche en temps logarithmique, au lieu de linéaire. L'algorithme se trouve ci-contre.

```
def recherche_dichotomique(val, tab):
    g = 0
    d = len(tab) - 1
    while g <= d:
        m = (g + d) // 2
        if tab[m] > val:
            d = m - 1
        elif tab[m] < val:
            g = m + 1
        else: # tab[m] == val
            return m
    return None
```

Exercice 2 : On considère ce tableau :

```
tab = [1, 3, 4, 7, 9, 11, 35, 79, 91, 135, 275, 478, 937, 1051, 3179]
```

1) Quelles sont les valeurs de `tab` qui sont regardées pour chercher `3` ?
2) Quelles sont les valeurs de `tab` qui sont regardées pour chercher `136` ?
3) Quelles sont les valeurs de `tab` qui sont regardées pour chercher `3179` ?

Exercice 3 : Combien de recherches faut-il faire, dans le pire des cas, pour trouver une valeur dans un tableau de longueur n, avec :

1) $n = 10$ 2) $n = 1\,000$ 3) $n = 1\,000\,000$

Terminaison

Le raison de la terminaison est assez intuitif. Puisqu'on divise par deux l'espace de recherche, qui contient un nombre fini de valeurs, on se retrouve rapidement avec une seule ou aucune valeur. Pour le justifier, il faut prendre "`d-g`" comme variant de boucle. À chaque tour de boucle, il y a 2 cas possibles. Soit on trouve le nombre, et c'est fini, soit on ne le trouve pas et dans ce cas, `g` augmente au moins de 1 ou `d` diminue au moins de 1. Si on n'est pas sorti de la boucle, $d-g$ diminue au moins de 1.
Dans le pire des cas, si le nombre cherché n'est pas dans le tableau, on se retrouve avec $d-g=-1$, ce qui fait sortir de la boucle. La terminaison est donc assurée.

Exercice 4 : Que se passe-t-il si le tableau donné est vide ?

Correction

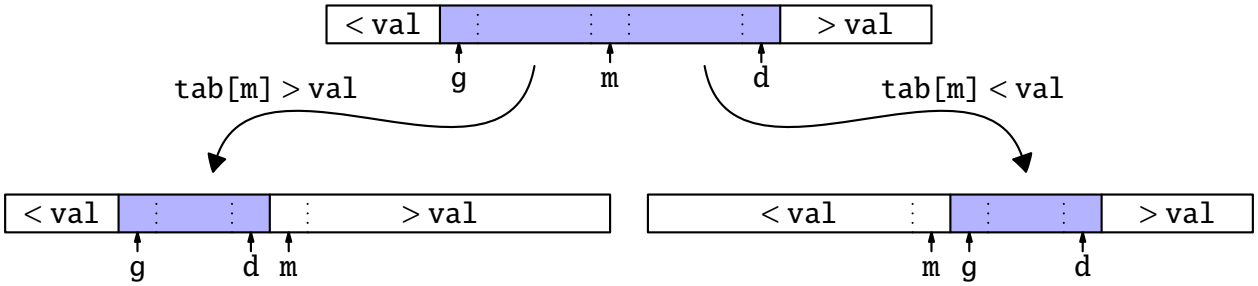
Pour s'assurer que le résultat est correct, il faut vérifier deux choses. Tout d'abord, si la fonction renvoie autre chose que `None`, c'est que `tab[m]` est bien égal au nombre cherché. Ce résultat est évident. Par contre, comment être sûr que si le nombre se trouve dans le tableau, la fonction trouvera une des positions où il se trouve, sans parcourir tout le tableau ?
Pour cela, nous allons utiliser l'invariant suivant :

Si `val` est dans `tab` alors $\texttt{tab[g]} \leq \texttt{val} \leq \texttt{tab[d]}$

Il y a 3 cas :

- Si $\texttt{tab[m]} = \texttt{val}$, alors on a trouvé une position contenant la valeur, ce qui est ce que l'on veut.
- Si $\texttt{tab[m]} < \texttt{val}$, alors on décale les valeurs vers la droite et on remplace `g` par `m+1`. Puisque le tableau est trié, si `val` est dans `tab`, c'est qu'il se trouve forcément après l'indice `m`. On a donc $\texttt{tab[m+1]} \leq \texttt{val} \leq \texttt{tab[d]}$.
- Si $\texttt{tab[m]} > \texttt{val}$, de façon similaire, on peut montrer que si `val` est dans `tab`, alors $\texttt{tab[g]} \leq \texttt{val} \leq \texttt{tab[m-1]}$.

On peut résumer cela par un schéma :



Donc si on n'a pas encore trouvé la valeur, l'invariant est préservé pour l'étape suivante. Puisque l'écart entre `g` et `d` diminue à chaque étape, si `val` est bien dans le tableau, on finira forcément par le trouver.
On en conclut donc que si la valeur est dans le tableau, on la trouve, et si la fonction renvoie une position, c'est que la valeur était bien dans le tableau. De plus, la fonction renvoie `None` si, et seulement si, la valeur ne se trouve pas dans le tableau.
L'algorithme est donc correct. Par contre, on peut remarquer que si une valeur est plusieurs fois dans le tableau, on ne sait pas quelle position sera renvoyée. Rien ne garantit que ce sera la première, comme dans le cas de l'algorithme naïf.