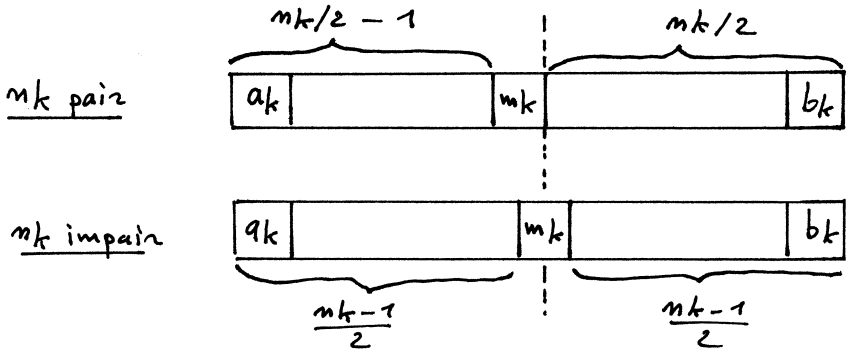
LYCÉE VAUGELAS 2021/2022	
MPSI	Informatique
Corrigé du TD Semaine 12	6 – 7/12/2021

Remarques générales sur la recherche dichotomique

- L'algorithme vous a été rappelé dans l'énoncé, mais la première des compétences à acquérir est d'être capable de le réécrire seul !
- L'étude de complexité doit être connue et elle est souvent à refaire dans les problèmes de concours.
- Les opérateurs arithmétiques, logiques, de comparaison et d'affectation doivent être précédés et suivis d'espaces dans le code Python. Les virgules doivent être suivies d'espace, mais jamais précédées d'espace. On tolère que le double point qui introduit un bloc ne soit pas précédé d'espace, même s'il s'agit d'un anglicisme.
- L'algorithme 1.2 est plus efficace en pratique si les comparaisons sont couteuses, car il en réalise deux fois moins.

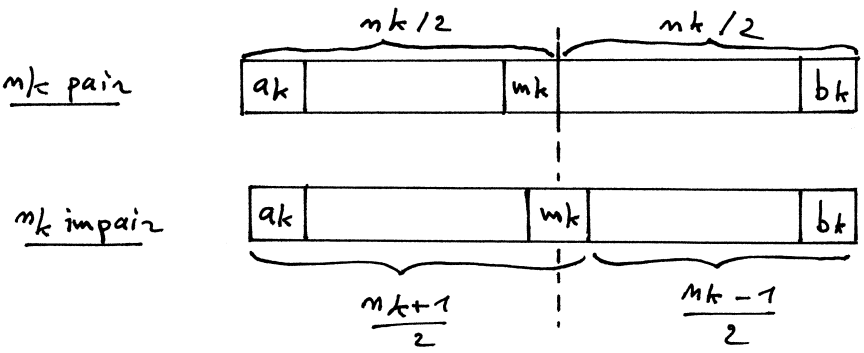
Exercice 1

1. $a_0 = 0$, $b_0 = n-1$, $n_o = n$, $m_0 = \left\lfloor \frac{n}{2} \right\rfloor$, c'est-à-dire $m_0 = n/2$ si n est pair et $m_0 = (n-1)/2$ si n est impair. De manière générale
$$n_k = b_k - a_k + 1 \tag{1}$$
où l'indice k numérote les itérations.
Les schémas doivent suggérer les relations, qu'il faut ensuite démontrer. Pour cela, on remarque que la parité de n_k est opposée à la parité de $b_k - a_k$ et à celle de $a_k + b_k = b_k - a_k + 2a_k$.
Algorithme 1.1



- Si n_k est pair : $m_k = a_k + \frac{n_k}{2} - 1 = \frac{a_k+b_k-1}{2}$.
  - Si $t[m_k] > x$, alors $n_{k+1} = \frac{n_k}{2} - 1$, $a_{k+1} = a_k$ et $b_{k+1} = m_k - 1 = \frac{a_k+b_k-3}{2}$
  - Si $t[m_k] \leqslant x$ alors $n_{k+1} = \frac{n_k}{2}$, $a_{k+1} = m_k + 1 = \frac{a_k+b_k+1}{2}$ et $b_{k+1} = b_k$.
- Si n_k est impair : $m_k = a_k + \frac{n_k-1}{2} = \frac{a_k+b_k}{2}$.
  - Si $t[m_k] > x$, alors $n_{k+1} = \frac{n_k-1}{2}$, $a_{k+1} = a_k$ et $b_{k+1} = m_k - 1 = \frac{a_k+b_k}{2} - 1$.
  - Si $t[m_k] \leqslant x$ alors $n_{k+1} = \frac{n_k-1}{2}$, $a_{k+1} = m_k + 1 = \frac{a_k+b_k}{2} + 1$ et $b_{k+1} = b_k$.

Algorithme 1.2



— Si n_k est pair : $m_k = a_k + \frac{n_k}{2} - 1 = \frac{a_k+b_k-1}{2}$.
  — Si $t[m_k] \geqslant x$, alors $n_{k+1} = \frac{n_k}{2}$, $a_{k+1} = a_k$ et $b_{k+1} = m_k = \frac{a_k+b_k-1}{2}$
  — Si $t[m_k] < x$ alors $n_{k+1} = \frac{n_k}{2}$, $a_{k+1} = m_k + 1 = \frac{a_k+b_k+1}{2}$ et $b_{k+1} = b_k$.
— Si n_k est impair : $m_k = a_k + \frac{n_k-1}{2} = \frac{a_k+b_k}{2}$.
  — Si $t[m_k] \geqslant x$, alors $n_{k+1} = \frac{n_k+1}{2}$, $a_{k+1} = a_k$ et $b_{k+1} = m_k = \frac{a_k+b_k}{2}$.
  — Si $t[m_k] < x$ alors $n_{k+1} = \frac{n_k-1}{2}$, $a_{k+1} = m_k + 1 = \frac{a_k+b_k}{2} + 1$ et $b_{k+1} = b_k$.

Conclusion pour l'algorithme 1.1 : on a dans tous les cas $n_{k+1} \leqslant \frac{n_k}{2}$. Par suite, comme $n_0 = n$, il vient

$$n_k \leqslant \frac{n}{2^k} \tag{2}$$

Terminaison de la recherche dichotomique

Nous avons en particulier montré la terminaison de l'algorithme 1.1, puisque l'inégalité $n_k \leqslant \frac{n}{2^k}$ montre que la condition $a_k \leqslant b_k$ finit par devenir fausse puisque $n_k = b_k - a_k + 1$. En d'autres termes, $b - a$ est un variant de boucle `while` et la fonction termine lorsque cet entier devient négatif. **Il suffisait d'ailleurs de remarquer la stricte décroissante de $b - a$ pour prouver la terminaison par un variant de boucle**, mais l'étude plus complète réalisée ici sera nécessaire pour prouver la complexité temporelle de l'algorithme.

Conclusion pour l'algorithme 1.2 : on a dans tous les cas $n_{k+1} \leqslant \frac{n_k+1}{2}$. Le point fixe de la suite récurrente vérifiant $n_{k+1} = \frac{n_k+1}{2}$ serait $f = 1$, donc on pose $u_k = n_k - 1$. Ainsi :

$$u_{k+1} = n_{k+1} - 1 \leqslant \frac{n_k - 1}{2} = \frac{u_k}{2}$$

Par suite $u_k \leqslant \frac{n-1}{2^k}$ car $u_0 = n - 1$. On a donc

$$n_k \leqslant \frac{n-1}{2^k} + 1 \leqslant \frac{n}{2^k} + 1 \tag{3}$$

Terminaison de la recherche dichotomique

Nous avons en particulier montré la terminaison de l'algorithme 1.2, puisque l'inégalité $n_k \leqslant \frac{n}{2^k} + 1$ montre que la condition $a_k < b_k$ finit par devenir fausse, car $n_k = b_k - a_k + 1$ implique $b_k - a_k \leqslant \frac{n}{2^k}$. En d'autres termes, $b - a$ est un variant de boucle `while` et la fonction termine lorsque cet entier devient nul. Ici encore, il aurait suffit de constater la stricte décroissance de $b - a$ pour prouver la terminaison (variant de boucle).

2. **Correction de l'algorithme 1.1**

La correction repose sur l'idée simple selon laquelle l'élément cherché ne peut se trouver que dans la partie de tableau que l'on conserve, qui repose précisément sur le fait que le tableau est trié. On utilise le théorème du cours : « Dans une boucle `while`, l'invariant de boucle est une propriété qui est vraie avant l'entrée dans la boucle et qui reste vraie après chaque parcours du corps de la boucle. »

On commence par constater que si l'élément x est absent du tableau t, alors la fonction retourne `False` car `y == x` n'est jamais vraie et que l'on a montré que la fonction termine. On suppose donc que $x \in t$. L'invariant de boucle est alors

$$t[a] \leqslant x \leqslant t[b] \qquad \text{et} \qquad x \in t[a : b + 1] \tag{4}$$

Cet invariant est vérifié à l'entrée dans la boucle puisque $a_0 = 0$ et $b_0 = n - 1$. Comme la boucle se termine lorsque $a > b$, l'élément x est toujours trouvé avant la fin de la boucle, car $x \in t$. Donc la fonction retourne `True`. La fonction est donc correcte dans tous les cas.

Correction de l'algorithme 1.2

Le principe selon lequel x appartient à la portion de tableau sur laquelle on poursuit la recherche reste inchangé. On a vu que la boucle termine toujours, il s'agit donc de démontrer que la valeur retournée `BOOL(T[a] = x)` est toujours la bonne. Il est clair que si x est absent du tableau, la valeur retournée est False puisque `t[a] = x` est faux quel que soit a. On suppose donc $x \in t$. L'invariant

$$a \leqslant b \qquad \text{et} \qquad x \in t[a : b+1] \tag{5}$$

montre alors la correction de l'algorithme car b contient la position de x dès qu'il est trouvé et $x \in t[a_k : b_k+1]$. Cet invariant montre que l'on sort de la boucle lorsque $a = b$ avec $x = t[a] = t[b]$.

Correction de l'algorithme 1.2

La correction de l'algorithme 1.2 repose donc sur le fait que l'on a toujours $a = b$ lorsqu'on sort de la boucle. En effet dès que x est trouvé, b n'évolue plus et conserve la position de x. De plus $m = \left\lfloor \frac{a+b}{2} \right\rfloor < b$ tant que $a < b$, donc $m + 1 \leqslant b$. Il est impossible que a dépasse b. On aurait pu tout aussi bien écrire le test `BOOL(T[b] = x)`.

3. Implémentation impérative de l'algorithme 1.1 en Python :

```
def recherche1(t, x):
    a, b = 0, len(t) - 1
    while a <= b:
        m = (a + b) // 2
        y = t[m]
        if y == x:
            return True
        if y > x:
            b = m - 1
        else:
            a = m + 1
    return False
```

4. Implémentation impérative de l'algorithme 1.2 en Python :

```
def recherche2(t, x):
    a, b = 0, len(t) - 1
    while a < b:
        m = (a + b) // 2
        if t[m] >= x:
            b = m
        else:
            a = m + 1
    return t[a] == x
```

Différence entre les deux algorithmes

L'avantage de cet algorithme 1.2 par rapport à l'algorithme 1.1 est de ne réaliser qu'une seule comparaison par itération, au lieu de deux pour l'algorithme 1.1. En contrepartie les itérations se poursuivent même si l'élément est trouvé. Comme on montre dans le cours que le nombre d'itérations en moyenne ne diffère que d'une unité du nombre d'itérations maximal, ce second algorithme est asymptotiquement deux fois plus rapide si la comparaison domine le temps de calcul.

5. L'implémentation récursive de l'algorithme 1.1 en Python est plus facile à appréhender que l'implémentation impérative puisqu'elle traduit exactement la nature récursive du principe dichotomique :

```python
def recherche3(t, x):
    def aux(a, b):
        if a > b:
            return False
        m = (a + b) // 2
        y = t[m]
        if y == x:
            return True
        if x < y:
            return aux(a, m - 1)
        return aux(m + 1, b)
    return aux(0, len(t) - 1)
```

6. Implémentation récursive de l'algorithme 1.2 en Python :

```python
def recherche4(t, x):
    def aux(a, b):
        if a >= b:
            return t[a] == x
        m = (a + b) // 2
        if x <= t[m]:
            return aux(a, m)
        return aux(m + 1, b)
    return aux(0, len(t) - 1)
```

7. On a montré pour l'algorithme 1.1 que

$$n_k \leqslant \frac{n}{2^k} \tag{6}$$

Comme $n_k = b_k - a_k + 1$, on a montré pour les bornes de l'intervalle de recherche

$$b_k - a_k \leqslant \frac{n}{2^k} - 1 \tag{7}$$

L'algorithme se termine au plus tard après la k-ième itération où k est le plus petit entier tel que $a_k > b_k$, d'où la condition suffisante

$$\frac{n}{2^k} < 1 \qquad \text{ou} \qquad k > \log_2 n \tag{8}$$

Le nombre d'itérations dans l'algorithme 1.1 de recherche dichotomique dans un tableau trié de n éléments est donc inférieur à $\lfloor \log_2 n \rfloor + 1$. Il s'agit du nombre de bits dans l'écriture binaire de n, ce qui est bien le résultat attendu puisque diviser n par 2 revient à décaler sa représentation binaire d'un bit vers la droite.

8. Si l'on remarque que le nombre d'opérations élémentaires réalisées à chaque itération est borné, on a montré que **la complexité temporelle de la recherche dichotomique est en $\mathbf{O}(\ln \mathbf{n})$.**
9. La complexité spatiale est en $\Theta(1)$ dans l'implémentation itérative et en $\Theta(\ln n)$ dans l'implémentation récursive (la récursivité terminale n'est pas dérécursifiée en Python).
10. Recherche dichotomique de la position d'un élément dans un tableau trié :

```python
def recherche5(t, x):
    def aux(a, b):
        if a >= b:
            return a if t[a] == x else -1
        m = (a + b) // 2
        if x <= t[m]:
            return aux(a, m)
        return aux(m + 1, b)
    return aux(0, len(t) - 1)
```

Fin