

DM2 - E3A MP 2019

1 Autour de la recherche par dichotomie

1.1 Question de cours

1. La recherche par dichotomie est un algorithme de recherche fonctionnant sur une liste d'entiers **triée**.

Le principe de cet algorithme consiste en la comparaison de l'élément recherché avec l'élément du milieu de la liste afin de savoir dans quel moitié de la liste chercher.

En procédant ainsi récursivement sur les moitiés désirées, cet algorithme à l'avantage d'avoir une complexité en $O(\ln n)$ où $n \in \mathbb{N}$ est la taille de la liste.

2. La recherche dichotomique nécessite que la liste soit triée, ce qui implique l'existence d'une relation d'ordre entre les éléments de la liste. Or il existe bien une relation d'ordre total pour les couples d'entiers et les chaînes de caractères (*ordre lexicographique*). La recherche par dichotomie est donc possible.

1.2 Étude d'une fonction dichotomie

3. Si on remplace la précondition de la ligne (3) par un appel à une fonction qui trierait la liste `liste` dans l'ordre croissant, alors soit ce tri modifierait directement la liste `liste`, ce qui menerait à des effets de bords, conséquence qui peut ne pas être souhaitable en fonction des besoins, soit l'on devra effectuer une copie de `liste`, et une complexité spatiale linéaire en la taille de la liste en résulterait, ce qui n'est pas souhaitable avec un algorithme de complexité temporelle logarithmique.

4. Soit $n \in \mathbb{N}$ la taille de la liste `liste`. Notons \mathcal{P}_t le prédicat : « l'entier x apparaît dans la sous-liste `liste[g:d+1]` en entrant dans le t -ème tour de boucle `while` ». Montrons par récurrence que $\mathcal{P}_1 = \mathcal{P}_t$ pour tout $t \in \mathbb{N}^*$:

Initialisation : si $t = 1$ (i.e. qu'on entre pour la première fois dans la boucle) alors on a évidemment $\mathcal{P}_1 = \mathcal{P}_t$

Hérédité : Soit $t \in \mathbb{N}^*$, supposons $\mathcal{P}_1 = \mathcal{P}_t$, montrons que $\mathcal{P}_1 = \mathcal{P}_{t+1}$: En entrant dans le t -ème tour de boucle :

- Soit $\mathcal{P}_1 = \mathcal{P}_t$ (hypothèse de récurrence) est vrai, et dans ce cas, pendant le t -ème tour de boucle :
 - Si x appartient à la moitié haute de la liste, alors au $(t+1)$ -ème tour de boucle, `liste[g, d + 1]` est égale à la moitié haute de la liste, donc \mathcal{P}_{t+1} est vrai et donc $\mathcal{P}_1 = \mathcal{P}_{t+1}$.
 - Sinon, x appartient à la moitié basse, et au tour de boucle $(t + 1)$, `liste[g, d + 1]` est égale à la moitié basse de la liste, donc \mathcal{P}_{t+1} est vrai et donc $\mathcal{P}_1 = \mathcal{P}_{t+1}$.
- Sinon, \mathcal{P}_1 est faux, et donc x n'apparaît pas dans `liste`, et n'apparaîtra donc jamais dans une sous-liste de `liste`. Ainsi $\mathcal{P}_1 = \mathcal{P}_t = \mathcal{P}_{t+1}$

On peut donc conclure que $\forall t \in \mathbb{N}^*$, $\mathcal{P}_1 = \mathcal{P}_t$, ce qui montre que \mathcal{P} est préservé à chaque tour de la boucle `while`

5. En prenant trois entiers $a < b < c \in \mathbb{Z}$, alors l'appel `dicho([a, c], b)` ne terminera pas.

6.


```
def dichotomie(liste,x):
    # Pré-conditions: x est un entier, liste est une
    # liste d'entiers triée dans l'ordre croissant
    n = len(liste)
    if n == 0:
        return False
    g, d = 0, n - 1
    while d - g > 0:
        m = (g+d)//2
        if liste[m] >= x:
            d = m
        else:
            g = m + 1
    return liste[g] == x
```

7. — Terminaison :

Montrons que la fonction termine, pour cela, montrons que la suite $(d - g)_{t \in \mathbb{N}^*}$ (la suite des valeurs de $d - g$ en rentrant de le t -ème tour de boucle) est strictement décroissante :

Soit $t \in \mathbb{N}^*$ tel que la boucle effectue le tour t (i.e. $(d - g)_t > 0$) :

- Si `liste[m] >= x`, d devient m et $m - g = (d - g)_{t+1} = \lfloor \frac{g+d}{2} \rfloor - g \leq \frac{g+d}{2} - g \leq \frac{d-g}{2} < (d - g)_t$
- Sinon, g devient $m + 1$, et $d - (m + 1) = (d - g)_{t+1} < d - \frac{g+d}{2} < \frac{d-g}{2} < (d - g)_t$

D'où la décroissance stricte et donc la terminaison.

— Correction :

De la même manière qu'à la question 4 on montre que \mathcal{P} est préservé à chaque tour de la boucle `while` de `dicho` corrigée, ce qui montre la correction.

1.3 Extensions du principe

8.

```
def tricho(liste:list, x:int) -> bool :
    n = len(liste)
    if n == 0:
        return False
    elif n == 1:
        return liste[0] == x
    else:
        t = n // 3
        if x <= liste[t]:
            return tricho(liste[0:t+1], x)
        elif x > liste[2 * t]:
            return tricho(liste[2*t + 1:n], x)
        else:
            return tricho(liste[t + 1: 2*t + 1], x)
```

9. La complexité de la recherche par dichotomie est en $O(\ln n)$, mais plus particulièrement, dans le pire des cas, elle est en $3 \log(n)_2$ (le facteur 3 vient du nombre de comparaison, et le log en base 2 vient du fait que l'on divise la liste en deux à chaque fois).

La complexité de la trichotomie est donc, dans le pire des cas, $4 \log(n)_3$. En notant C_d et C_t les complexités respectives des deux algorithmes de recherche, on a alors

$$C_t = 4 \frac{\ln n}{\ln 3} = \frac{4}{3} \cdot \frac{\ln 2}{\ln 3} \cdot \frac{3 \ln n}{\ln 2} = \frac{4}{3} \cdot \frac{\ln 2}{\ln 3} \cdot C_d \approx 0.84 \cdot C_d$$

Ainsi $C_d \approx 1.2 C_t$, et la complexité asymptotique reste donc la même : $O(\ln(n))$

10.

```
def dicho_matrice(mat:list, x:int) -> tuple :
    n = len(mat)
    p = len(mat[0])
    if n == 0 or p == 0:
        return False
    d_row, f_row = 0, n - 1
    d_column, f_column = 0, p - 1
    while f_column - d_column > 0:
        mid_column = (d_column + f_column) // 2
        if x <= mat[n - 1][mid_column]:
            f_column = mid_column
        else:
            d_column = mid_column + 1
    while f_row - d_row > 0:
        mid_row = (d_row + f_row) // 2
        if x <= mat[mid_row][d_column]:
            f_row = mid_row
        else:
            d_row = mid_row + 1
    if mat[d_row][d_column] == x:
```

```

    return (d_row, d_column)
else:
    return (-1, -1)

```

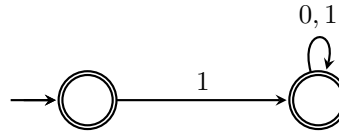
2 Automates et langages de mots binaires

11. a) $41 = 1010001_2$

b) $10101010_2 = 88$

c) Un automate est local lorsque pour chaque lettre σ , toutes les transitions étiquetées par σ arrivent dans un même état. Un automate est dit standard lorsqu'il n'existe aucune transition aboutissant à l'état initial.

d)



Automate \mathcal{A}_1

e)

```

let langage_1 = function
| [] -> true
|h::_ -> h ;;

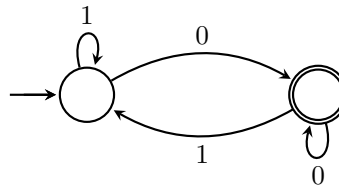
```

12. a) Avec $R = \{0, 1\}$, $S = \{0\}$, alors

$$\begin{aligned}
 (RA^* \cap A^*S) \setminus (A^* \underbrace{\emptyset}_{\in A^2} A^*) &= (A^* \cap A^*S) \setminus \emptyset \\
 &= A^*S \\
 &= L_2
 \end{aligned}$$

Donc L_2 est bien un langage local.

b)



Automate \mathcal{A}_2

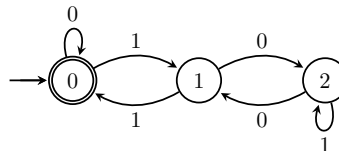
c)

```

let rec langage_2 = function
| [] -> false
|h::[] -> not h
|_::t -> langage_2 t ;;

```

13. a)



Automate \mathcal{A}_3

b) Montrons que $\mathcal{P}(n)$ est vrai pour tout $n \in \mathbb{N}^*$:

Initialisation : si $n = 1$, $\forall a \in A$, $\delta^*(0, a) = \delta(0, a) = (2 \cdot 0 + a) \bmod 3 = (a \cdot 2^{1-1}) \bmod 3$.

Hérédité : Soit $n \in \mathbb{N}^*$, supposons $\mathcal{P}(n)$, montrons $\mathcal{P}(n) \implies \mathcal{P}(n+1)$:

Soit m un mot de A de taille $n + 1$, on peut écrire $m = w_1 \dots w_{n+1}$ où $\forall i \in \llbracket 1, n + 1 \rrbracket, w_i \in A$.

$$\begin{aligned}
 \text{Ainsi : } \delta^*(0, m) &= \delta^*(0, w_1 \dots w_{n+1}) \\
 &= \delta(\delta^*(0, w_2 \dots w_{n+1}), w_1) && \text{par définition de } \delta^* \\
 &= 2 \left[\left(\sum_{k=2}^{n+1} w_k 2^{n-k} \right) \bmod 3 \right] + w_1 \bmod 3 && \text{par l'hypothèse de récurrence} \\
 &= \sum_{k=1}^{n+1} w_k 2^{n+1-k} \bmod 3
 \end{aligned}$$

Ainsi on montre bien $\mathcal{P}(n)$ pour tout $n \in \mathbb{N}^*$

14. a) Le langage L_1 correspond à tous les entiers naturels en base 2.
 Le langage L_2 correspond à tous les entiers naturels pairs en base 2.
 La question 13b. montre que L_3 correspond à tous les entiers naturels multiples de 3 en base 2.
Ainsi $L_4 = L_1 \cap L_2 \cap L_3$ correspond à tous les entiers naturels à la fois multiple de 2 et 3, c'est-à-dire à tous les entiers naturels multiples de 6 en base 2.
- b) Les langages L_1, L_2, L_3 sont clairement reconnaissable (on a exhiber un automate les reconnaissant), donc, comme intersection de langages reconnaissables, L_4 est reconnaissable.
D'où l'existence d'un automate reconnaissant L_4 .

3 Diamètre d'un graphe

3.1 Exemples de graphes

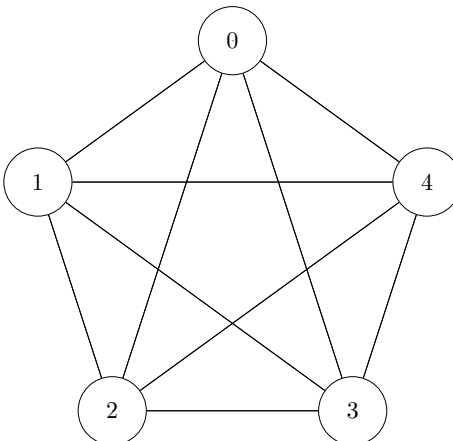
15. diam(G_1) = 3 ; chemins maximaux : (0 - 2 - 3 - 4), (0 - 2 - 3 - 5), (1 - 2 - 3 - 4), (1 - 2 - 3 - 5), (4 - 3 - 2 - 0), (5 - 3 - 2 - 0), (4 - 3 - 2 - 1), (5 - 3 - 2 - 1).
- diam(G_2) = 3 ; chemins maximaux : (2 - 1 - 0 - 5), (1 - 0 - 5 - 4), (0 - 5 - 4 - 3), (5 - 4 - 3 - 2), (4 - 3 - 2 - 1), (3 - 2 - 1 - 0), (0 - 1 - 2 - 3), (1 - 2 - 3 - 4), (2 - 3 - 4 - 5), (3 - 4 - 5 - 0), (4 - 5 - 0 - 1), (5 - 0 - 1 - 2).

16. a) 

Graphe à 5 sommets de diamètre maximum

- b)

```
let diam_max n = (*cree un graphe "lineaire"*)
  let g = Array.make n [] in
  if n > 1 then (g.(0) <- [1]; g.(n - 1) <- [n - 2]) ;
  for i = 1 to n - 2 do
    g.(i) <- [i + 1; i - 1]
  done;
  g ;;
```

17. a) 

Graphe à 5 sommets de diamètre minimum

```

b) let diam_min n = (*cree un graphe complet*)
    let g = Array.make n [] in
    for i = 0 to n - 1 do
        for j = 0 to n - 1 do
            if i <> j then g.(i) <- j::g.(i)
        done;
    done;
    g ;;

```

3.2 Algorithmes de calcul du diamètre

18. L'algorithme de Dijkstra prend en entrée un graphe pondéré avec des poids positifs, et un sommet de départ. La sortie de l'algorithme est alors les plus courts chemins depuis le sommet de départ vers chacun des autres sommets du graphe.

En pondérant toutes les arêtes avec des poids de 1, et en appliquant l'algorithme de Dijkstra à chacun des sommets du graphe, on pourrait alors connaître le chemin le plus court de poids maximal, et ce poids sera la diamètre du graphe.

19. Le parcours en largeur peut être utilisé pour calculer le diamètre d'un graphe, en effet en le faisant partir depuis chacun des sommets du graphe, alors on peut connaître toutes les distances, et donc la maximum.
20. On note $n, p \in \mathbb{N}$ respectivement le nombre de sommets et d'arêtes du graphe alors la complexité de Dijkstra est au mieux $O(n + p \log p)$ (elle dépend en fait de la technique utilisée pour trouver un minimum dans un ensemble de poids). La complexité d'un parcours en largeur est quant à elle $O(n + p)$. De plus dans les deux cas, on effectue l'algorithme en partant de chacun des sommets, soit n fois.

L'algorithme du parcours en largeur paraît donc plus adapté pour calculer le diamètre d'un graphe.

3.3 Diamètre d'un arbre binaire

```

21. let a = Noeud(0,
    Noeud(1,
        (Noeud(2,
            Noeud(4, Feuille, Feuille),
            Feuille)),
        Noeud(3,
            Noeud(5, Feuille, Feuille),
            Noeud(6, Feuille, Feuille))),
    Feuille) ;;

```

$\text{diam}(\mathcal{A}) = 4$; chemins maximaux : $(4-2-1-3-6)$, $(6-3-1-2-4)$, $(4-2-1-3-5)$, $(5-3-1-2-4)$

22. Notons $\mathcal{P}(n)$ le prédicat : « le graphe sous-jacent d'un arbre à n sommets possède $n-1$ arêtes ». Montrons que $\mathcal{P}(n)$ est vrai pour tout $n \in \mathbb{N}^*$:

Initialisation : si $n = 1$, alors le graphe sous-jacent est réduit à un unique sommet, et il possède $0 = n - 1$ arête.

Hérédité : Soit $n \in \mathbb{N}^*$, supposons $\mathcal{P}(n)$, montrons $\mathcal{P}(n) \implies \mathcal{P}(n+1)$:

Soit \mathcal{G} le graphe sous-jacent d'un arbre binaire à n sommets. Le nombre d'arêtes de \mathcal{G} est alors $n - 1$ (hypothèse de récurrence).

Ainsi, en adjoignant à \mathcal{G} un nouveau sommet (on peut l'appeler n sans perdre de généralité, quitte à renommer les sommets de \mathcal{G}), par définition d'un arbre binaire, il n'est relié qu'à un seul autre sommet.

Ainsi $\mathcal{G} + \{n\}$ est le graphe sous-jacent d'un arbre binaire à $n + 1$ sommets et possède n arêtes.

Donc $r = n - 1$

```

23. let rec nb_noeuds = function
    | Feuille -> 0
    | Noeud(_, g, d) -> 1 + nb_noeuds g + nb_noeuds d ;;

```

24.

```

let numerotation arbre =
  let index = ref (-1) in
  let rec aux = function
    |Feuille -> Feuille
    |Noeud(_, g, d) -> (incr index ;
                        let i = !index in Noeud(i, aux g, aux d))
  in
  aux arbre ;;

```

25.

```

let arbre_vers_graphe arbre =
  let graphe = Array.make (nb_noeuds arbre) [] in
  let rec aux sommet = function
    |Feuille -> ()
    |Noeud(a, g, d) -> if sommet <> -1 (*la racine n'a pas d'antecedant*)
                        then begin
                           graphe.(a) <- sommet::graphe.(a) ;
                           graphe.(sommet) <- a::graphe.(sommet)
                         end;
                        aux a g;
                        aux a d
  in
  aux (-1) arbre ;
  graphe ;;

```

26. On peut calculer le diamètre d'un arbre en le transformant en graphe et effectuant des parcours en largeur comme énoncé à la question 19. Si l'arbre n'est pas correctement numéroté, on peut utilisé `numerotation`.

Soit n le nombre de noeuds d'un arbre. La complexité de `nb_noeuds` est $O(n)$, celle de `numerotation` est aussi $O(n)$. Enfin le parcours en largeur effectué n fois a pour complexité $O(n(n + n - 1)) = O(n^2)$.

La complexité totale de l'algorithme est donc $O(n^2)$

27. Soit \mathcal{A} un arbre binaire, si \mathcal{A} est vide ou réduit à une feuille, alors il n'existe pas de chemin (graphe sous-jacent vide). Sinon on note $\mathcal{A} = (r, \mathcal{A}_g, \mathcal{A}_d)$. On pose $h(\text{Feuille}) = -1$ La longueur maximal d'un chemin passant par la racine est alors :

$$\underbrace{1 + 1}_{\text{distance pour aller à } \mathcal{A}_g \text{ et } \mathcal{A}_d} + \overbrace{h(\mathcal{A}_g) + h(\mathcal{A}_d)}^{\text{par définition de } h}$$

28.

```

let rec diam_arbre arbre =
  let rec aux = function (*renvoie diametre, hauteur*)
    |Feuille -> (0, -1) (*on pose diam(Feuille) = 0 pour des raisons de
    ↪ cohérence*)
    |Noeud(_, g, d) -> let diam_g, haut_g = aux g in
                        let diam_d, haut_d = aux d in
                        let max_hauteur = 1 + max haut_g haut_d in
                        let chemin_racine = 2 + haut_g + haut_d in
                        let max_diam_fils = max diam_g diam_d in
                        (max chemin_racine max_diam_fils), max_hauteur
  in
  fst (aux arbre) ;;

```

Récursivement, la fonction `aux` ne parcourt bien qu'une seule fois chacun des noeuds.

La complexité de `diam_arbre` est donc bien linéaire en le nombre de noeuds de l'arbre.