version pdf

Synthèse du cours Diviser pour Régner

Recherche dichotomique

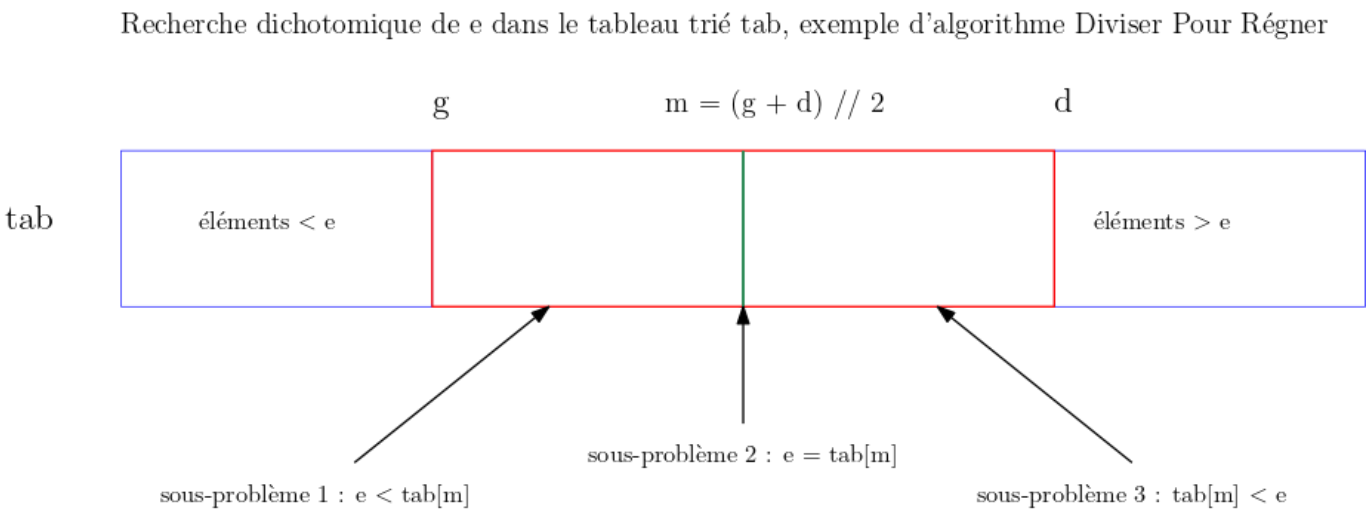
"Point de cours 1 : recherche dichotomique dans un tableau trié"

Spécification du problème :

- On dispose d'un tableau `tab` d'éléments de même type et compararables. De plus le tableau est trié dans l'ordre croissant et les indices commencent à 0.
- On nous donne un élément `e` du même type que ceux du tableau.
- On veut rechercher une occurrence de `e` dans le tableau et renvoyer son indice. Si `e` n'est pas dans le tableau, on renverra -1.

L'algorithme de **recherche dichotomique** exploite l'ordre sur les éléments pour diviser au moins par deux la taille de la zone de recherche à chaque étape. On note `g` et `d` les indices délimitant à gauche et à droite la *zone de recherche*. On initialise `g` avec 0 et `d` avec la longueur du tableau moins 1. Ensuite on répète les étapes suivantes jusqu'à ce que l'on trouve une occurrence de `e` ou que la *zone de recherche* soit vide (cas où `e` n'est pas dans le tableau).

- **Diviser :** On calcule l'indice du milieu de la zone de recherche `m = (g + d) // 2` et on se ramène à la résolution de trois sous-problèmes plus petits et similaires :



- **Résoudre :** On résout directement le sous-problème 1 si `tab[m] = e` et sinon on résout récursivement l'un des deux autres sous-problèmes :
  - Si `e < tab[m]` , sous-problème 2 : l'élément `e` ne peut être que dans la première moitié de la zone de recherche correspondant aux indices dans l'intervalle `[g, m[`
  - Si `tab[m] < e` , sous-problème 3 : l'élément `e` ne peut être que dans la seconde moitié de la zone de recherche correspondant aux indices dans l'intervalle `]m, d]`

"Implémentation itérative"

```
def recherche_dicho_iter(t, e):
    """
    Recherche dichotomique dans un tableau trié dans l'ordre croissant
    Paramètres :
        t : tableau d'éléments  de même type et comparables, précondit
        e : un élément du même type que ceux  dans tab
    Retour:
        Si e dans tab renvoie l'index d'une occurrence sinon renvoie -
    """
    g, d = 0, len(t) - 1
    while g <= d:
        m = (g + d) // 2
        # Sous-problème 1 :  occurrence de e en t[m]
        if e == t[m]:
            return m
        # Sous-problème 2 :  occurrence de e en t[m]
        #  on continue la recherche dans la première moitié [g, m [ =
        elif e < t[m]:
            d = m - 1
        # Sous problème 3
        # on continue la recherche dans la seconde moitié [m + 1, d]
        else:
            g = m + 1
    return -1
```

"Implémentation récursive"

```python
def recherche_dicho_rec(t, e, g, d):
    m = (g + d) // 2
    # 1er cas de base : zone de recherche vide
    if g > d:
        return -1
    # Sous-problème 1
    # 2eme cas de base :  occurrence de e en t[m]
    if t[m] == e:
        return m
    # Sous problème 2
    # on continue la recherche dans la première moitié [g, m [ = [g, m - 1]
    elif e < t[m]:
        return recherche_dicho_rec(t, e, g, m - 1)
    # Sous problème 3
    # on continue la recherche dans la seconde moitié [m + 1, d]
    else:
        return recherche_dicho_rec(t, e, m + 1, d)

def recherche_dicho_rec_env(t, e):
    """Fonction enveloppe"""
    return recherche_dicho_rec(t, e, 0, len(t) - 1)
```

"Point de cours 2"

La recherche dichotomique dans un tableau trié de taille n est beaucoup plus efficace en nombre de comparaisons que la recherche séquentielle.

Algorithme de recherche dans un tableau trié	Complexité temporelle dans le pire des cas	Equivalent
Recherche dichotomique (Diviser pour Régner)	logarithmique $O(\log_2(n))$	Nombre de chiffres de n en base 2
Recherche séquentielle	linéaire $O(n)$	Valeur de n

Méthode *Diviser pour Régner* {#méthode-*diviser-pour-régner* }

"Point de cours 3"

La méthode *Diviser Pour Régner* [1] est une généralisation de la dichotomie.

Un algorithme de type *Diviser pour Régner* est caractérisé par trois étapes :

1. **Diviser :** étant donné le problème à résoudre, on découpe l'entrée en deux ou plusieurs *sous-problèmes similaires, plus petits et indépendants*.
2. **Résoudre :** on résout tous les sous-problèmes :
   - soit directement si la solution est simple (*cas de base*)
   - soit en appelant l'algorithme sur le sous-problème (*appel récursif*)
3. **Combiner :** à partir des solutions des sous-problèmes on reconstitue une solution du problème initial [2].

La réduction d'un problème à des sous-problèmes similaires et plus petits, conduit naturellement à une programmation récursive.

"Remarque"

La méthode *Diviser pour Régner* permet parfois d'améliorer la complexité en temps, comme pour la recherche d'un élément dans un tableau trié, mais ce n'est pas toujours le cas. Par exemple, l'algorithme *Diviser pour Régner* de recherche du maximum dans un tableau d'entiers a une complexité linéaire, comme la recherche séquentielle.

"Recherche de maximum DpR"

```python
def maximum_dpr(tab):
    # Cas de base des appels récursifs : sous-problème de résolution direct
    if len(tab) == 1:
        return tab[0]
    # Diviser
    m = len(tab) // 2
    # Résoudre les 2 sous-problèmes
    m1 = maximum_dpr(tab[:m])
    m2 = maximum_dpr(tab[m:])
    # Combiner les solutions des sous-problèmes
    if m1 >= m2:
        return m1
    return m2
```

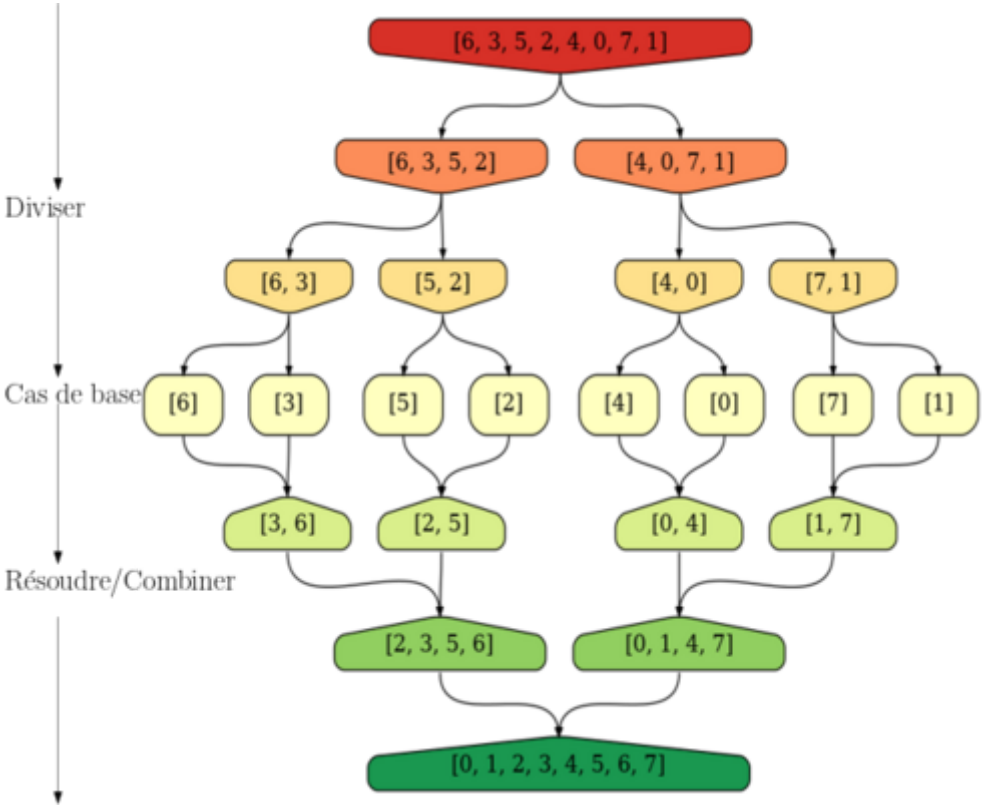
Tri fusion

"Point de cours 3 : tri fusion"

L'algorithme de **tri par fusion** permet de trier un tableau `t` d'éléments comparables avec une méthode *Diviser pour Régner* :

1. **Diviser :** on découpe le tableau en son milieu `m` et on se ramène à deux sous-problèmes similaires et plus petits :
   - trier le premier sous-tableau avec les éléments d'indice inférieur ou égal à `m`
   - trier le second sous-tableau avec les éléments d'indice supérieur à `m`
2. **Résoudre :** on résout les deux sous-problèmes en appelant récursivement l'algorithme sur chaque sous-tableau et on obtient deux sous-tableaux triés `t1` et `t2`.
3. **Combiner :** on fusionne les deux sous-tableaux triés `t1` et `t2` en un tableau trié `t3` contenant les mêmes éléments que `t`.

"exemple"



On a représenté par le schéma ci-dessus, la trace d'exécution de l'algorithme de **tri fusion** qui trie dans l'ordre croissant le tableau d'entiers `[6, 3, 5, 2, 4, 0, 7 , 1]` .

"Implémentation récursive du tri fusion"

```
def tri_fusion(t):
    """Renvoie un tabbleau avec les mêmes éléments que t tableau d'entiers
    # cas de base:
    if len(t) <= 1:
        return t
    # diviser
    m = len(t) // 2
    # résoudre les sous-problèmes
    t1 = tri_fusion(t[:m])
    t2 = tri_fusion(t[m:])
    # combiner
    t3 = fusion(t1, t2)
    return t3
```

"Implémentation itérative de la fonction fusion"

```python
def fusion(t1, t2):
    """
    Fusionne les tableaux d'entiers  t1 et t2 triés dans l'ordre croissant
    """
    n1 = len(t1)
    n2 = len(t2)
    t3 = []
    i1, i2 = 0, 0
    while i1 < n1 and i2 < n2:
        if t1[i1] <= t2[i2]:
            t3.append(t1[i1])
            i1 = i1 + 1
        else:
            t3.append(t2[i2])
            i2 = i2 + 1
    # ici un des deux tableaux t1 ou t2 est vide
    # cas il reste t1 non vide
    while i1 < n1:
        t3.append(t1[i1])
        i1 = i1 + 1
    # cas il reste t2 non vide
    while i2 < n2:
        t3.append(t2[i2])
        i2 = i2 + 1
    return t3
```

"Remarque"

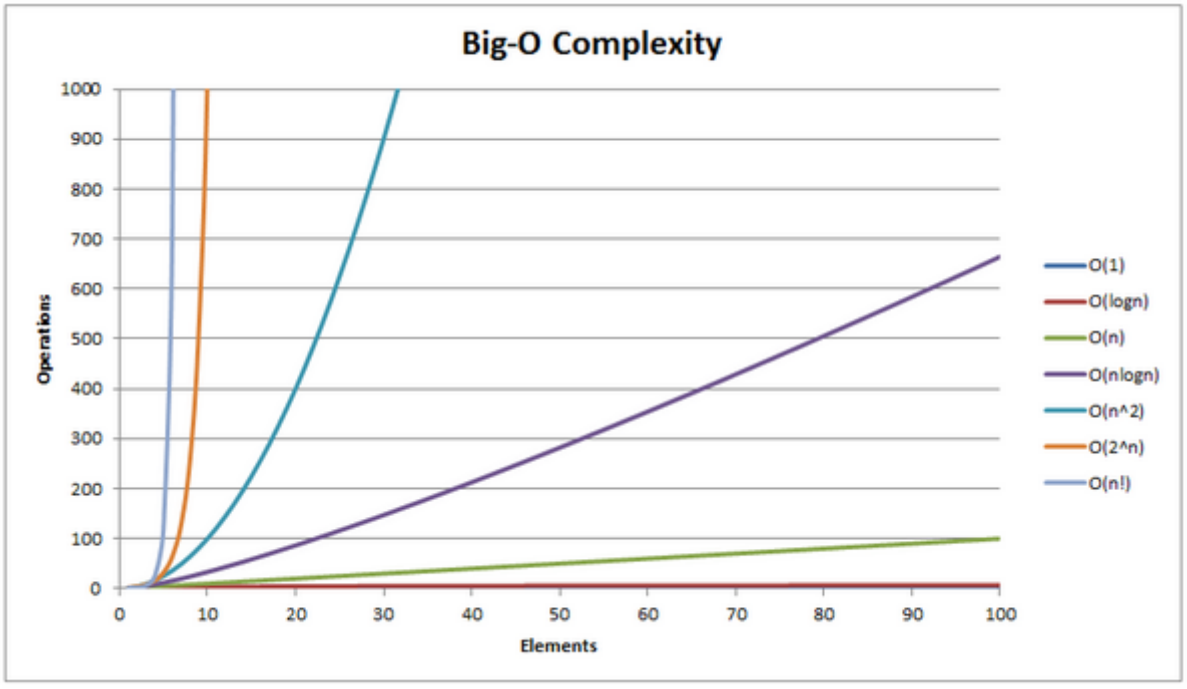
Cette version n'effectue pas de tri en place en redistribuant les éléments dans le tableau initial, mais il est possible d'implémenter un *tri fusion* en place avec la même complexité en temps en utilisant un tableau auxiliaire de stockage pour les fusions.

"Point de cours 4"

La complexité en temps du **tri fusion** d'un tableau de taille n est *linéarithmique*, en $O(n \log_2(n))$.
Cette complexité est *optimale pour les tris par comparaison* de deux éléments. Le *tri fusion* est

plus efficace que les algorithmes de tri vus en classe de première qui sont de complexité *quadratique*, en $O(n^2)$. Le *tri rapide* est un autre algorithme *Diviser pour Régner*, très efficace. L'implémentation, plus délicate, sera vue en TP.

Algorithme de tri d'un tableau de taille n	**Complexité dans le meilleur des cas**	**Complexité dans le cas moyen**	**Complexité dans le pire des cas**
tri par sélection	$O(n^2)$	$O(n^2)$	$O(n^2)$
tri par insertion	$O(n)$ (tableau déjà trié)	$O(n^2)$ (ordre inverse)	$O(n^2)$
tri par bulles	$O(n^2)$	$O(n^2)$	$O(n^2)$
tri fusion	$O(n\log_2(n))$	$O(n\log_2(n))$	$O(n\log_2(n))$
tri rapide	$O(n\log_2(n))$	$O(n\log_2(n))$	$O(n^2)$ (tableau déjà trié)



> *Source :* https://www.hackerearth.com/practice/notes/sorting-and-searching-algorithms-time-complexities-cheat-sheet/

1. *Divide and Conquer* en anglais ↩

2. L'étape **Combiner** est absente de la *recherche dichotomique* mais bien présente dans d'autres exemples vus en cours comme la recherche de maximum *Diviser pour Régner* ou le *tri fusion*. ↩