Introduction à l'informatique

pour les mathématiques, la physique et les sciences computationnelles

Yann Thorimbert

Partie 2 | Chapitre 3
Algorithmes de recherche au sein d'une séquence

Yann Thorimbert

UNIVERSITÉ DE GENÈVE
CENTRE UNIVERSITAIRE D'INFORMATIQUE

Chapitres du cours (seconde partie du cours)

0. Introduction

1. Structures de données : tableaux, listes et dictionnaires

2. Algorithmes de tri

3. Algorithmes de recherche au sein d'une séquence ←

4. Algorithmes sur graphes

Algorithmes de recherche au sein d'une séquence

Algorithmes de recherche | **Séquences quelconques**

- But : La séquence S contient la quantité *q*. Il faut trouver l'indice auquel ce nombre se trouve.

- Dans le cas général où les éléments sont dans un ordre quelconque, il faut parcourir la séquence jusqu'à trouver celui qui nous intéresse :

```
1. Pour chaque entier i de 0 à n-1:
    a. Si S[i] = q, retourner i
```

- La complexité est alors $O(n)$.

Algorithmes de recherche | **Séquences ordonnées**

- Si la séquence est ordonnée, on peut exploiter l'ordonnancement pour améliorer la performance de la recherche.

- La recherche dichotomique est un exemple intuitif d'un algorithme exploitant une telle propriété.

- Spontanément, bon nombre d'humains mettent en oeuvre une sorte de recherche dichotomique lorsqu'on leur demande de deviner un nombre.

Jeu de la devinette

- Jeu : le joueur doit deviner un entier entre 1 et 100. Quand il se trompe, on lui indique si le nombre à trouver est plus petit ou plus grand que ce qu'il a tenté.

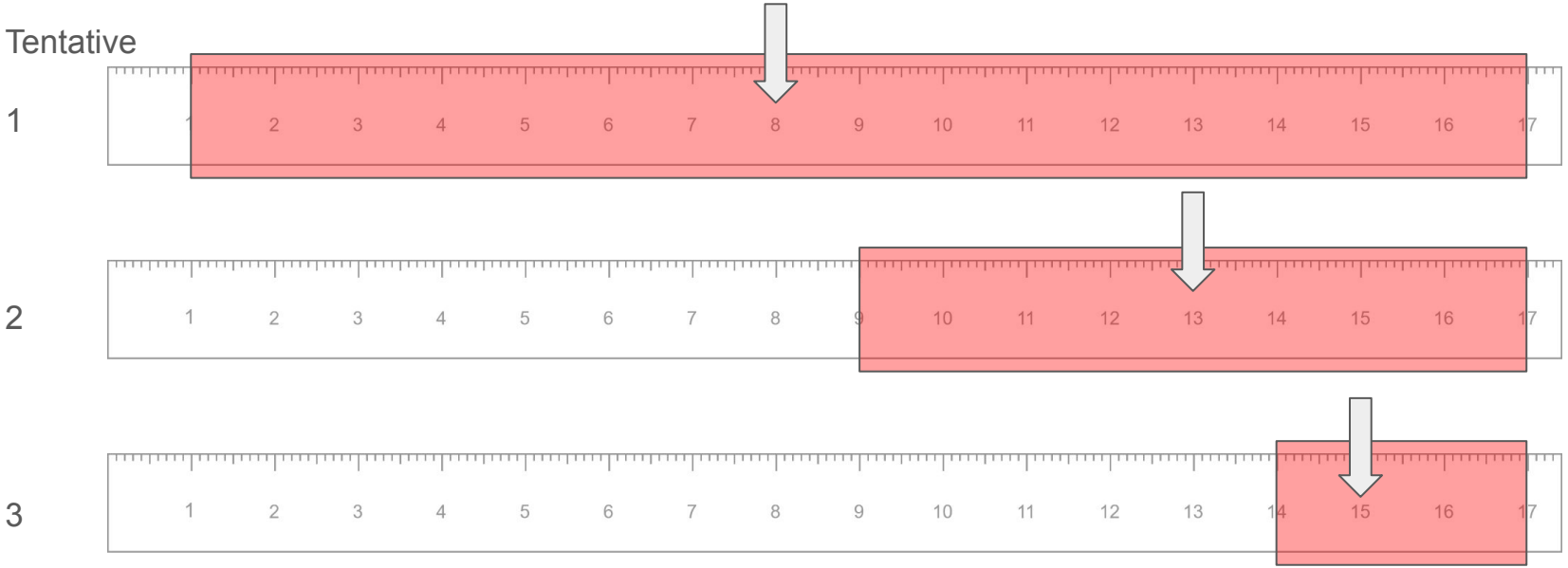
- Exemple d'historique de partie :

```
-Le nombre vaut-il 50 ?
-Plus petit.
-Vaut-il 25 ?
-Plus grand ?
-Vaut-il 38 ?
(etc.)
```

À chaque fois, le joueur tente la valeur au milieu de l'intervalle des entiers non-éliminés.

Jeu de la devinette

À chaque fois, le joueur tente la valeur au milieu de l'intervalle des entiers non-éliminés.



Jeu de la devinette

Avec cet algorithme, comment évolue la probabilité de trouver le bon nombre au prochain coup en fonction du nombre de tentatives k ?

- À chaque tentative, l'intervalle I des nombres non-éliminés diminue de moitié :
  $\Rightarrow I_k$ est de taille $I_0 / 2^k$ à l'étape k.
- Si on en est à notre tentative numéro k, alors le prochain nombre tenté aura une probabilité $1 / I_k = 2^k / I_0$ d'être le bon.
- Autrement dit : la probabilité de trouver le bon nombre à la prochaine tentative augmente exponentiellement avec le nombre de tentatives.
- Dans le pire des cas, on n'aura plus qu'un nombre possible après $k = \log_2(I_0)$ étapes.

Recherche dichotomique | **Idée générale**

1. On compare le nombre cherché avec le nombre au milieu de la séquence.

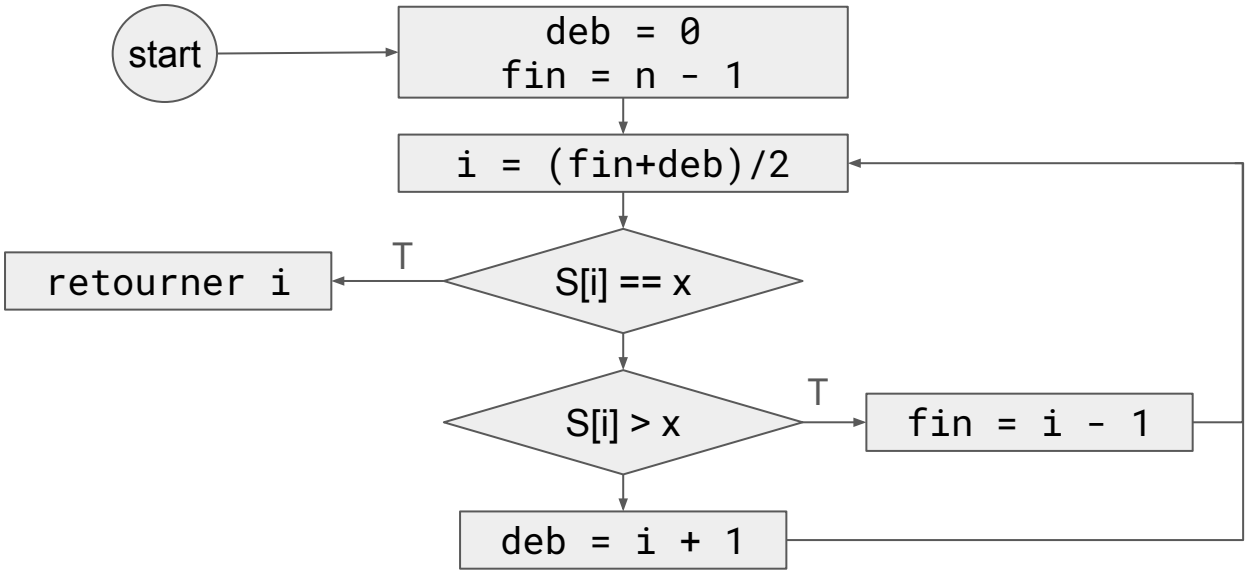
2. Si le nombre cherché est plus petit que le nombre au milieu de la séquence, on revient au point 1 après avoir décidé que la nouvelle séquence à considérer est la première moitié de la séquence initiale.

3. Sinon, si le nombre cherché est plus grand que le nombre au milieu de la séquence, on revient au point 1 après avoir décidé que la nouvelle liste à considérer est la deuxième moitié de la séquence initiale.

4. Sinon, on a trouvé l'emplacement du nombre cherché.

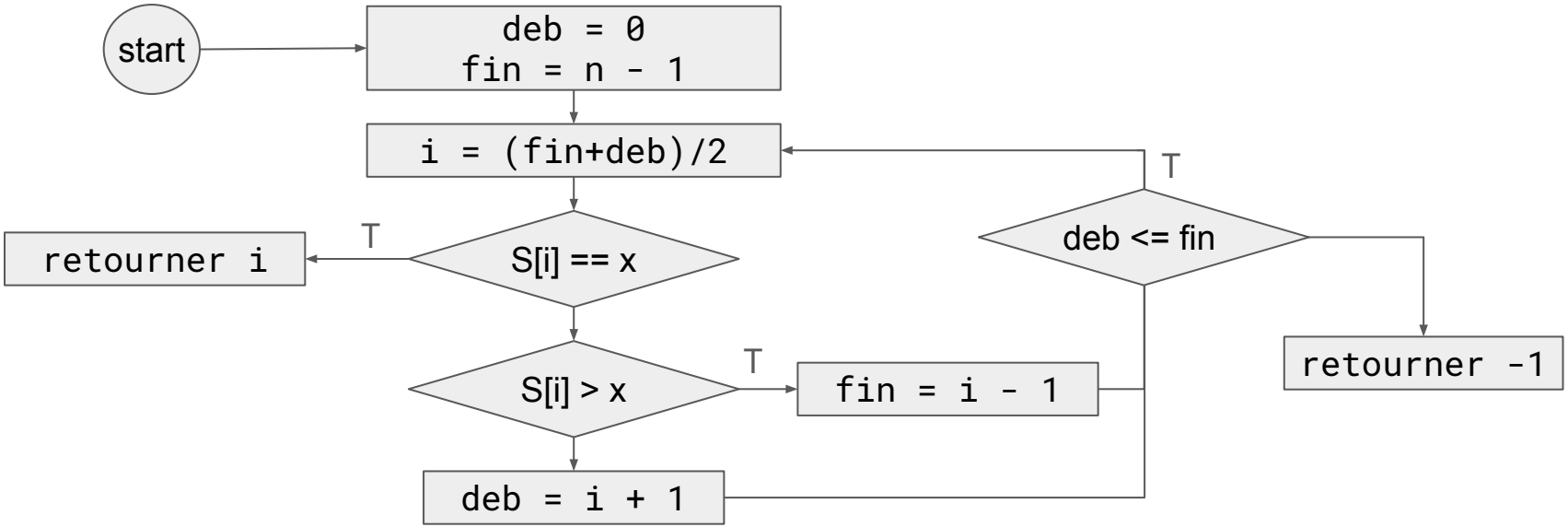
Recherche dichotomique | **Algorigramme**

Version où l'on suppose que x fait partie de S



Recherche dichotomique | **Algorigramme**

Version complète où l'on retourne -1 si x ∉ S



Recherche dichotomique | **Commentaires sur la division**

- Que faire de l'instruction `i = (fin + deb) / 2` ?
- i représente l'indice du milieu du tableau, c'est donc un entier.
- Exemple 1: si `fin = 4` et `deb = 2`, alors `i = (4+2)/2 = 3` (ok).
- Exemple 2 : si `fin = 5` et `deb = 2`, alors `i = (5+2)/2 = 3.5` (pas ok).
- Solution : **arrondir** soit vers le bas, soit vers le haut. Une façon commune de procéder est d'utiliser une division entière (arrondi vers le bas).
  - Exemple en Python : `i = (fin + deb) // 2`
  - `Exemple en MATLAB : i = floor((fin + deb)/2);`

Recherche dichotomique | **Commentaires sur x ∉ S**

- D'après la façon dont notre algorithme est formulé, l'indice de début ne peut qu'augmenter et l'indice de fin ne peut que diminuer.

- La taille de la sous-séquence à considérer vaut `fin - debut`.

- Cela n'a pas de sens si la taille est négative, c'est à dire si `fin < debut`. C'est là une condition pour déterminer que l'élément ne figure pas dans la séquence.

Recherche dichotomique | **Exemple pas à pas**

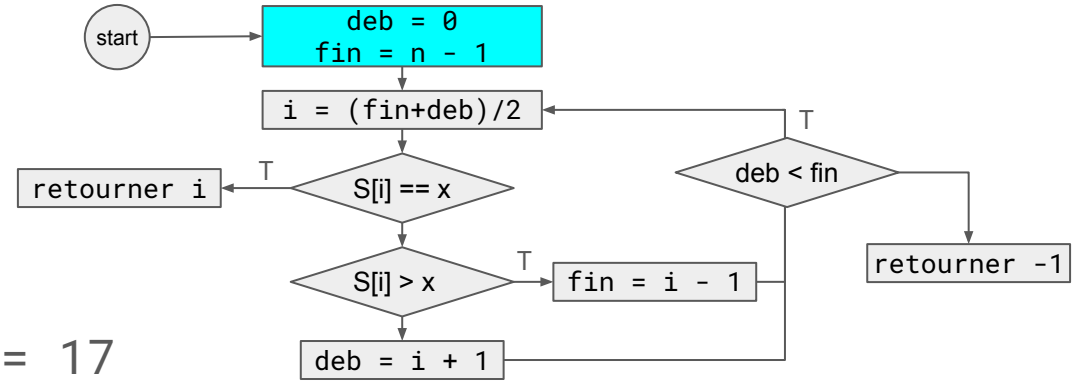
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
3	3	4	6	7	8	8	11	13	17	18	20	22	24	25	26

On cherche x = 17

Recherche dichotomique | **Exemple pas à pas**

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
3	3	4	6	7	8	8	11	13	17	18	20	22	24	25	26

deb **fin**

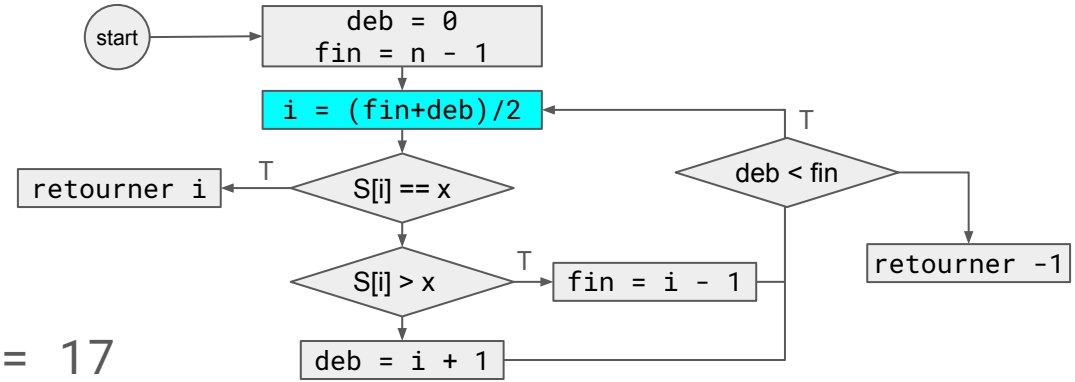


On cherche x = 17

Recherche dichotomique | **Exemple pas à pas**

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
3	3	4	6	7	8	8	11	13	17	18	20	22	24	25	26

deb i fin

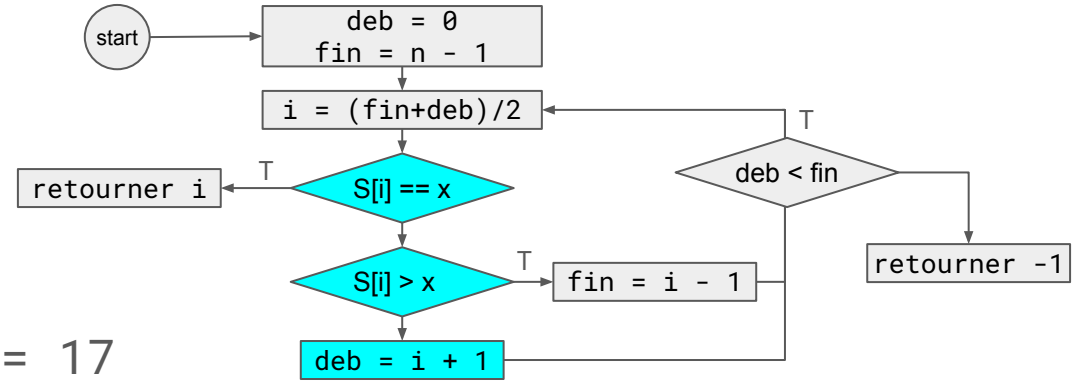


On cherche x = 17

Recherche dichotomique | **Exemple pas à pas**

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
3	3	4	6	7	8	8	11	13	17	18	20	22	24	25	26

i deb fin

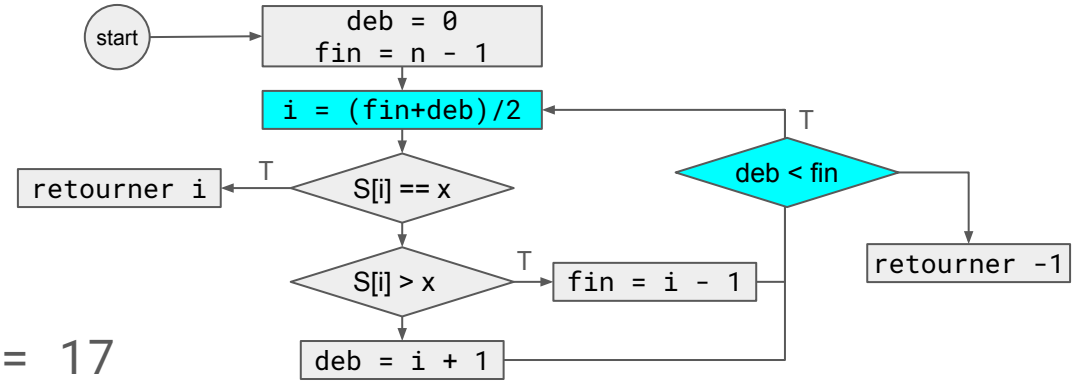


On cherche x = 17

Recherche dichotomique | **Exemple pas à pas**

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
3	3	4	6	7	8	8	11	13	17	18	20	22	24	25	26

deb i fin

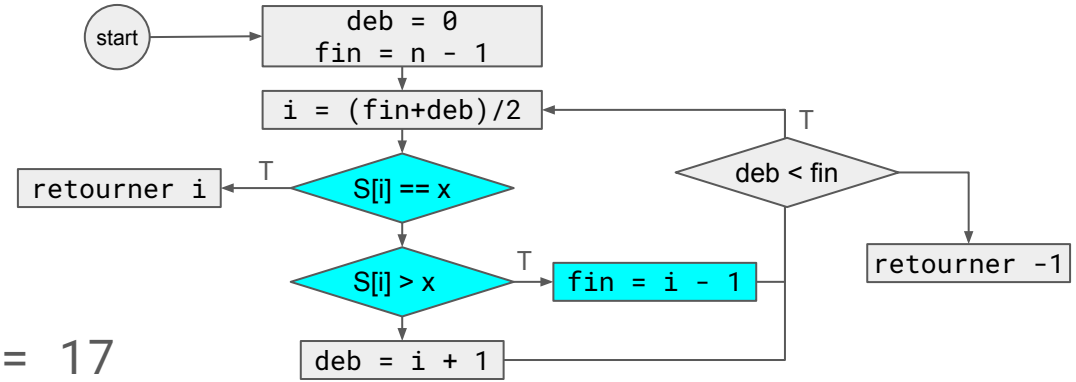


On cherche x = 17

Recherche dichotomique | **Exemple pas à pas**

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
3	3	4	6	7	8	8	11	13	17	18	20	22	24	25	26

deb fin i

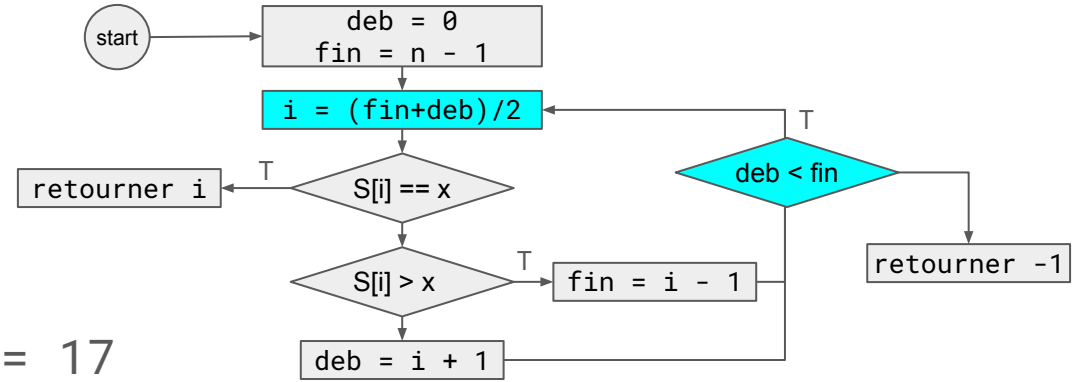


On cherche x = 17

Recherche dichotomique | **Exemple pas à pas**

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
3	3	4	6	7	8	8	11	13	17	18	20	22	24	25	26

deb i fin

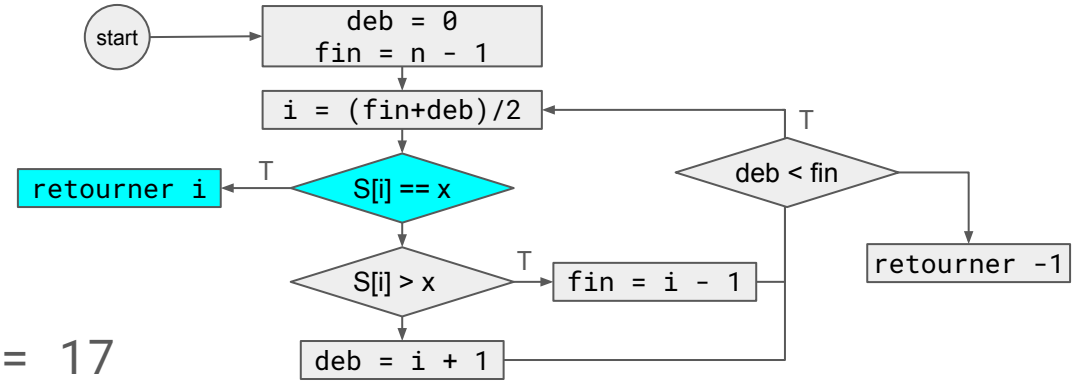


On cherche x = 17

Recherche dichotomique | **Exemple pas à pas**

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
3	3	4	6	7	8	8	11	13	17	18	20	22	24	25	26

i



On cherche x = 17

Question

Quelle est la complexité en temps de la recherche dichotomique ?

votamatic.unige.ch

code ZXNX

Question

Quelle est la complexité en temps de la recherche dichotomique ?

Réponse : elle est logarithmique, car on doit au pire diviser la liste $\log_2(n)$ fois pour arriver à un seul élément. À chaque fois, on réalise un nombre constant d'étapes pour vérifier si l'élément du milieu est le bon.

Rappel : "combien de fois peut-on diviser n par 2 avant d'obtenir 1 ?" correspond à l'équation $n / 2^k = 1$, d'où $k = \log2(n)$.

Implémentation

Notez que deux versions sont possibles :

- version **itérative** → Laissée en exercice.
- version **récursive** → À voir maintenant.

Implémentation en Python de la version récursive

```
def recherche_dichotomique(a, v, debut, fin):
    if debut > fin:
        return -1
    milieu = (debut + fin) // 2
    if a[milieu] < v:
        return recherche_dichotomique(a, v, milieu + 1, fin)
    elif a[milieu] > v:
        return recherche_dichotomique(a, v, debut, milieu - 1)
    else:
        return milieu
```

Recherche de l'élément majoritaire

Recherche de l'élément majoritaire

- Soit une séquence de nombres S = [3, 5, 2, 1, 9, …].

- On cherche l'élément m qui constitue **plus de la moitié des éléments**, s'il existe. Autrement dit, le nombre d'occurrences de l'élément m est strictement supérieur à n // 2.

- m se nomme l'**élément majoritaire** de la séquence.
  Par exemple, si S = [3, 5, 3, 1, 3], alors m = 3 est l'élément majoritaire.

- Idée d'approche naïve ?

Élément majoritaire | **Approche naïve**

- Parcourir n fois la liste :
  - Une première fois pour compter le nombre d'occurrences du premier élément.
  - Une seconde fois pour compter le nombre d'occurrences du second élément.
  - Etc. À chaque fois, si le nombre d'occurrences est supérieur à n / 2, on a trouvé.

- Complexité $O(n^2)$

- Comment faire mieux ?

Élément majoritaire | **Approche "Diviser pour régner"**

- L'idée est de former n/2 couples de valeurs.
- Si m est majoritaire, il devrait apparaître dans la moitié des couples au moins.
- On peut réduire les couples pour former une séquence S', qui garde le même élément majoritaire (attention : quelques rectifications à faire cependant).
- Si un couple est hétérogène, il se réduit au néant, sinon il se réduit à l'élément répété.
- Exemple : $S = [\underbrace{3, 1}, \underbrace{3, 3}, \underbrace{2, 1}, \underbrace{3, 3}]$

  $S' = [\ \varnothing, \quad 3, \quad \varnothing, \quad 3] = [3, 3]$

Élément majoritaire | **Approche "Diviser pour régner"**

- La taille de la séquence à vérifier est divisée par 2 à chaque fois jusqu'à n'obtenir qu'un élément : $n / 2^k = 1$, et donc le nombre d'étapes k est logarithmique.

- Cependant, à chaque étape, il faut effectuer n comparaisons si la séquence est de longueur n.

- Le nombre d'opérations à effectuer est donc $n + n/2 + n/4 + \ldots + 1 = 2n - 1$.
  $\Rightarrow$ L'algorithme semble $O(n)$ dans le pire des cas.

Élément majoritaire | **Approche "Diviser pour régner"**

- Problème : si n est impair, que faire de la dernière valeur ?

  `S = [3, 1, 3, 3, 2, 3, 1], que faire du 1 ?`

- Le garder ne fonctionne pas, car ne préserve pas l'élément majoritaire.

  `S = [3, 1, 3, 3, 2, 3, 1]`

  `S' = [ ∅, 3, ∅, 1] = [3, 1]`

- L'ignorer ne fonctionne pas non plus :

  `S = [1, 3, 1, 3, 3]`

  `S' = [ ∅, ∅, ∅ ] = [ ]`

Élément majoritaire | **Approche "Diviser pour régner"**

- Problème : si n est impair, que faire de la dernière valeur ?
  S = [3, 1, 3, 3, 2, 3, 1] ← que faire du 1 ?

- La solution est de tester si l'élément en trop est majoritaire, et de le jeter s'il ne l'est pas.

- L'opération est *O*(n), et comme il y a log(n) étapes, on obtient finalement une complexité dans le pire des cas de *O*(n log(n)).

- Cela dit, s'il y a un élément majoritaire, l'algorithme est susceptible de le trouver très vite.

Élément majoritaire | **Approche "Diviser pour régner"**

Voici le principe de l'algorithme (ne fonctionne pas tel quel !) :

```
def element_majoritaire(a):
    """a est la sous-séquence considérée."""
    n = len(a)
    nouvelle_sequence = [] # séquence à utiliser pour la prochaine récursion.
    for i in range(0, n-1, 2):
        if a[i] == a[i+1]: # ajout de l'élément si couple homogène
            nouvelle_sequence.append(a[i])
    return element_majoritaire(nouvelle_sequence) # appel récursif.
```

(Diapositive hors champ)

Élément majoritaire | **Approche "Diviser pour régner"**

```python
def element_majoritaire(S, a):
    """S est la séquence de base, a est la sous-séquence considérée."""
    n = len(a)
    debut = 0 # contient l'indice du debut à considérer (cf. "élément en trop")
    if n == 0:
        return None # si la liste est vide, pas d'élément majoritaire
    elif n%2==1: # si la taille est impaire, on vérifie le premier élément.
        if compte(S, a[0]) > len(S) // 2:
            return a[0] # l'élément "en trop" est majoritaire, fin de l'algo.
        debut = 1 # si la taille est impaire, on ignorera le premier élément.
    nouvelle_sequence = [] # séquence à utiliser pour la prochaine récursion.
    for i in range(debut, n-1, 2):
        if a[i] == a[i+1]: # ajout de l'élément si couple homogène
            nouvelle_sequence.append(a[i])
    return element_majoritaire(S, nouvelle_sequence) # appel récursif.
```

(Diapositive hors champ)

Élément majoritaire | **Approche par dictionnaire**

- Pourquoi ne pas tenir un compteur d'occurrences pour chaque valeur, au fure et à mesure qu'on parcourt le tableau ?

- Le dictionnaire est idéal pour cela : les clés sont les éléments de S, et les valeurs sont le nombre d'occurrences de ces clés.

- Nommons d le dictionnaire. Par exemple, d[3] contient le nombre d'occurrences de 3 au sein de S.

Élément majoritaire | **Approche par dictionnaire**

```python
def element_majoritaire_dict(a):
    n = len(a)
    d = {}
    for e in a: #O(n)
        if e in d: #O(1) en moyenne, O(n) dans le pire des cas
            d[e] += 1 #O(1)
        else:
            d[e] = 1 #O(1)
        if d[e] > n // 2: #O(1)
            return e
    return None
```

(Version MATLAB : Cf. corrigé série 13)

Élément majoritaire | **Conclusion**

- L'approche divide-and-conquer donne la meilleure complexité dans le pire des cas.

- En pratique, l'approche par dictionnaire est très efficace (à tester pour soi !).