Trier et rechercher

INFORMATIQUE COMMUNE - TP nº 1.4 - Olivier Reynet

À la fin de ce chapitre, je sais :

- ☞ coder un algorithme de tri simple et explicité
- ☞ évaluer le temps d'exécution d'un algorithme avec la bibliothèque time
- ☞ rechercher un élément dans un tableau séquentiellement ou par dichotomie itérative
- ☞ générer un graphique légendé avec la bibliothèque matplotlib

A Trier un tableau

A1. On souhaite trier des listes Python, considérées ici comme des tableaux, avec des algorithmes différents (cf. algorithmes 1, 2 et 3). Chaque algorithme de tri est implémenté par une fonction Python. Le prototype de ces fonctions est `my_sort(t)`, où `t` est un paramètre formel qui représente le tableau à trier.

```
def my_sort(t):
  # tri du tableau
  # for i in range(len(t))
  #     t[i] = ...
```

Cette fonction, une fois réalisée, trie le tableau `t` passé en paramètre mais ne renvoie rien (i.e. pas de `return`). Expliquer pourquoi.

Solution : Une liste Python est une variable muable, donc on peut modifier `t` sans pour autant changer la référence de `t` en mémoire. Les modifications apportées par l'algorithme sur `t` le sont directement sur la liste passée en paramètre effectif par référence (toute variable Python étant une référence vers une case mémoire). C'est pourquoi on n'a pas besoin d'effectuer un `return` pour récupérer le travail de la fonction.

A2. Coder les algorithmes de tri par sélection, par insertion et par comptage en respectant le prototype défini à la question précédente[1].

Solution :

```
def selection_sort(t):
  for i in range(len(t)):
    min_index = i
```

1. On a le droit de collaborer, de se répartir les algorithmes et de s'échanger les codes s'ils sont corrects !

```python
        for j in range(i + 1, len(t)):
            if t[j] < t[min_index]:
                min_index = j
        swap(t, i, min_index)


def insertion_sort(t):
    for i in range(1, len(t)):
        to_insert = t[i]
        j = i
        while t[j - 1] > to_insert and j > 0:
            t[j] = t[j - 1]
            j -= 1
        t[j] = to_insert


def counting_sort(t):
    v_max = max(t)
    count = [0] * (v_max + 1)
    for e in t:
        count[e] += 1
    output = [None for i in range(len(t))]
    i = 0  # indice de parcours du tableau résultat
    for v in range(v_max + 1):
        for j in range(count[v]):
            output[i] = v
            i += 1
    return output
```

A3. Tester ces algorithmes sur une **même** liste Python de longueur 20 et contenant de types `int` choisis aléatoirement entre 0 et 100.

Solution :

```python
N = 20
M = 100

t = [randint(0, M) for _ in range(N)]
copy_t = t[::]  # création d'une copie du tableau t
                # pas comme copy_t = t, référence seulement
selection_sort(copy_t)
print(copy_t)
copy_t = t[::]
insertion_sort(copy_t)
print(copy_t)
copy_t = t[::]
counting_sort(copy_t)
print(copy_t)
```

A4. Peut-t-on trier des listes de chaînes de caractères avec ces mêmes codes ? Tester cette possbilité à l'aide de la liste `["Zorglub", "Spirou", "Fantasio", "Marsupilami", "Marsu", "Samovar", "Zantafio"]`. Analyser les résultats. Pourquoi est-ce possible ? Pourquoi n'est-ce pas possible ?

Solution : Les tris par sélection ou par insertion sont des tris comparatifs : ils utilisent la comparaison de deux éléments du tableau pour trier les éléments. Lorsque les éléments sont des entiers, cela fonctionne grâce à l'ordre sur les entiers naturels. Lorsque les éléments sont des chaînes de caractères, c'est l'ordre lexicographique qui est utilisé. Cet ordre s'appuie sur l'ordre alphabétique et la position du caractère dans la chaîne pour comparer deux chaînes. Python utilise implicitement l'ordre lexicographique lorsque on compare deux chaînes de caractères avec l'opérateur <. C'est pourquoi les tris par insertion, sélection ou bulles fonctionnent aussi dans ce cas.

Par contre, le tri par comptage n'est pas un tri comparatif : c'est un tri par dénombrement de valeurs entières. Il ne porte donc que sur des tableaux contenant des entiers et ne peut pas fonctionner pour des chaînes de caractères.

Algorithme 1 Tri par sélection

```
Fonction TRIER_SELECTION(t)
    n ← taille(t)
    pour i de 0 à n − 1 répéter
        min_index ← i                                  ▷ indice du prochain plus petit
        pour j de i + 1 à n − 1 répéter                ▷ pour tous les éléments non triés
            si t[j] < t[min_index ] alors
                min_index ← j                          ▷ c'est l'indice du plus petit non trié!
        échanger(t[i], t[min_index])                   ▷ c'est le plus grand des triés!
```

Algorithme 2 Tri par insertion

```
Fonction TRIER_INSERTION(t)
    n ← taille(t)
    pour i de 1 à n-1 répéter
        à_insérer ← t[i]
        j ← i
        tant que t[j-1] > à_insérer et j>0  répéter
            t[j] ← t[j-1]                              ▷ faire monter les éléments
            j ← j-1
        t[j] ← à_insérer                               ▷ insertion de l'élément
```

La bibliothèque `matplotlib` permet de générer des graphiques à partir de données de type `list` qui constituent les abscisses et les ordonnées associées. La démarche à suivre est de :

- importer la bibliothèque `from matplotlib import pyplot as plt`
- créer une figure `plt.figure()`
- tracer une courbe `plt.plot(x,y)` si `x` et `y` sont les listes des abscisses et des ordonnées associées. La bibliothèque trace les points `(x[i], y[i])` sur le graphique.
- ajouter les éléments de légende et de titre,
- montrer la figure ainsi réalisée `plt.show()`.

Algorithme 3 Tri par comptage

Fonction TRIER_COMPTAGE(t, v_{max}) ▷ v_{max} est le plus grand entier à trier
    n ← taille(t)
    c ← un tableau de taille $v_{max}+1$ initialisé avec des zéros
    pour i de 0 à $n-1$ **répéter**
        c[t[i]] ← c[t[i]] + 1 ▷ compter les occurrences de chaque élément du tableau.
    résultat ← un tableau de taille n
    $i \leftarrow 0$
    pour v de 0 à v_{max} **répéter** ▷ On prend chaque valeur possible dans l'ordre
        si c[v] > 0 **alors** ▷ Si l'élément v est présent dans le tableau
            pour j de 0 à c[v] - 1 **répéter**
                résultat[i] ← v ▷ alors écrire autant de v que d'occurrences de v
                $i \leftarrow i+1$ ▷ à la bonne place, la ième !
    renvoyer résultat

La bibliothèque `time` permet notamment de mesurer le temps d'exécution d'un code. Un exemple de code utilisant ces deux bibliothèques est donné ci-dessous. Le graphique qui en résulte est montré sur la figure 1.

Code 1 – Example d'utilisation des bibliothèques time et matplotlib

```python
import time
from matplotlib import pyplot as plt


def to_measure(d):
    time.sleep(d)  # Do nothing, wait for d seconds


# Simple use
tic = time.perf_counter()
to_measure(0.1)
toc = time.perf_counter()

print(f"Execution time : {toc - tic} seconds")

# Plotting results
timing = []
delay = [d / 1000 for d in range(1, 100, 5)]
for d in delay:
    tic = time.perf_counter()
    to_measure(d)
    toc = time.perf_counter()
    timing.append(toc - tic)

plt.figure()
plt.plot(delay, timing, color='cyan', label='fonction to_measure')
plt.xlabel('Delay', fontsize=18)
plt.ylabel("Execution time", fontsize=16)
plt.legend()
plt.show()
```

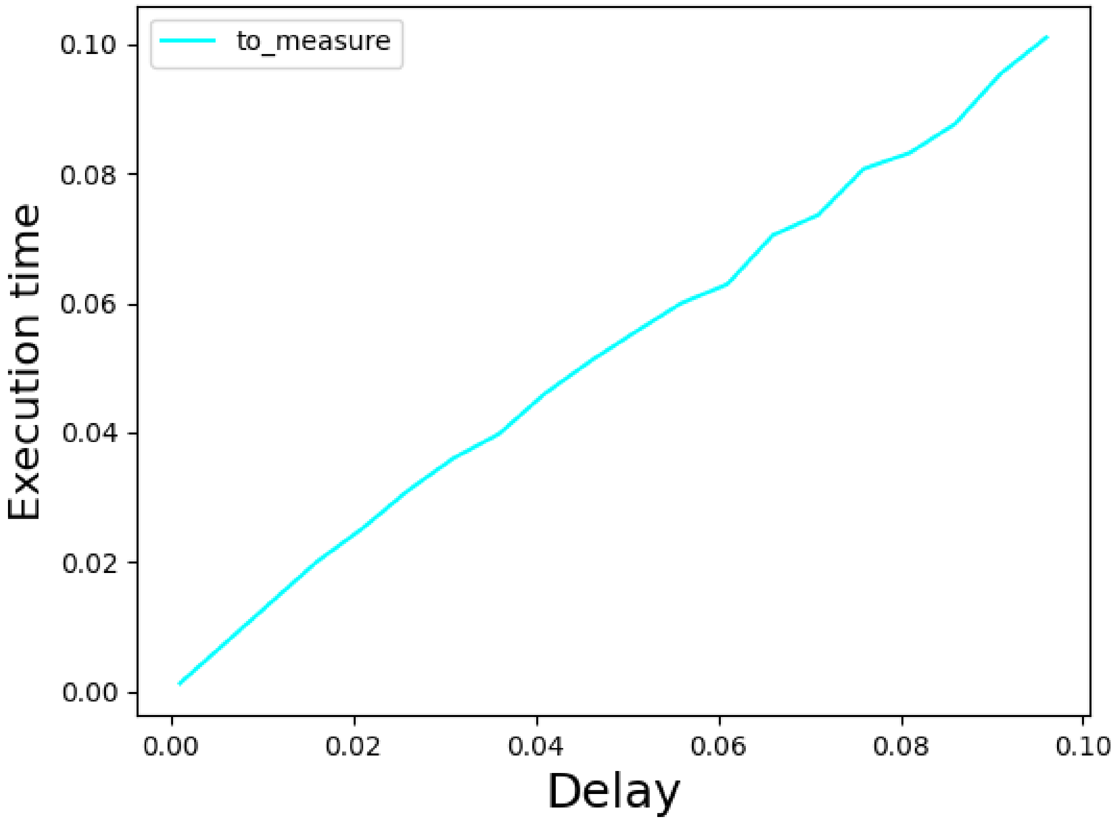


FIGURE 1 – Figure obtenue à partir des bibliothèques matplotlib et time et du code 1

A5. À l'aide de la bibliothèque matplotlib, tracer les temps d'exécution nécessaires au tri d'un même tableau d'entiers par les algorithmes implémentés. On pourra également les comparer à la fonction `sorted` de Python. Analyser les résultats. Essayer de qualifier les coûts des algorithmes en fonction de la taille du tableau d'entrée.

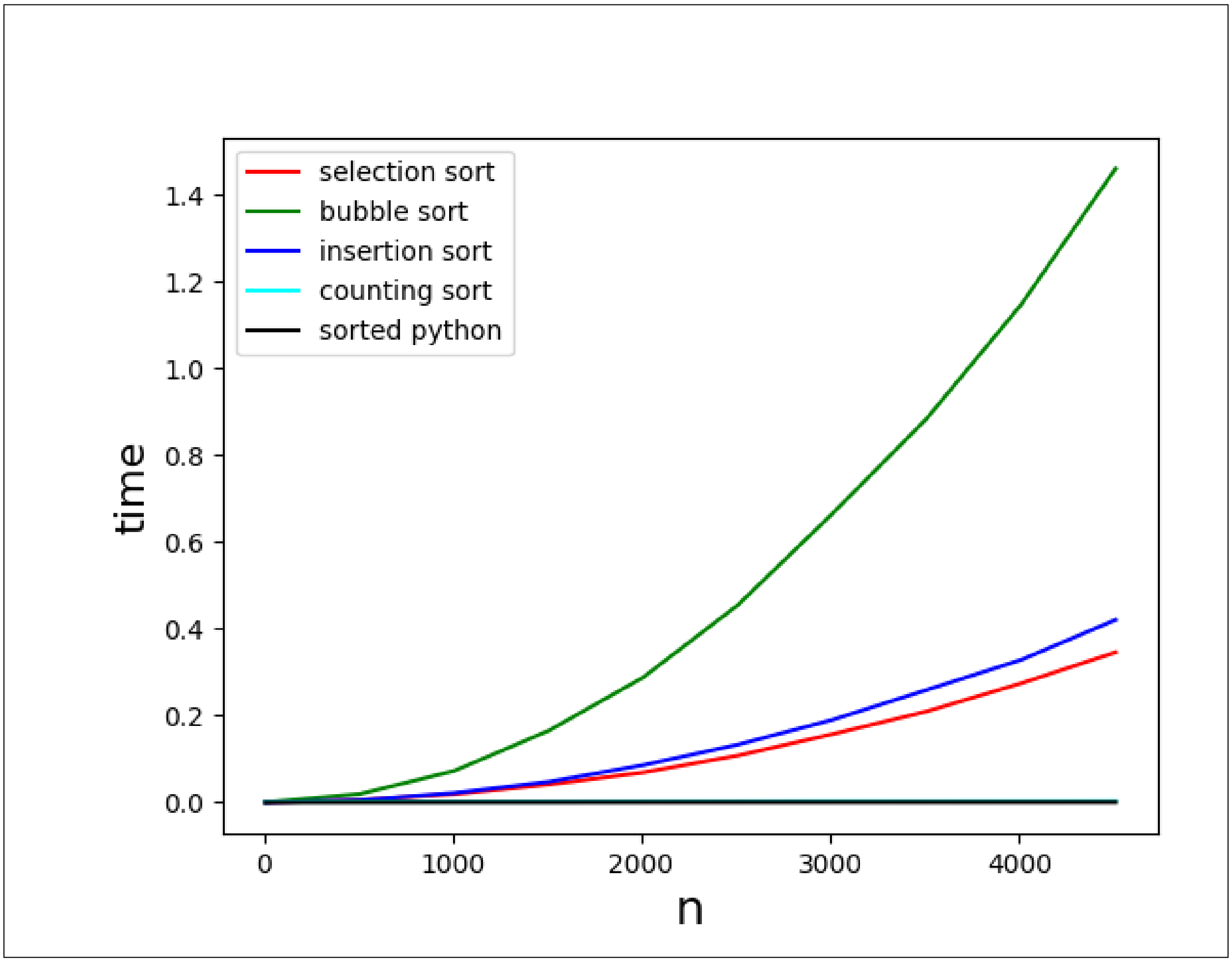
Solution :

Code 2 – Trier des tableaux

```
import time
from random import randint


def swap(t, i, j):
    t[i], t[j] = t[j], t[i]


def selection_sort(t):
    for i in range(len(t)):
        min_index = i
        for j in range(i + 1, len(t)):
            if t[j] < t[min_index]:
                min_index = j
        swap(t, i, min_index)


def insertion_sort(t):
    for i in range(1, len(t)):
        to_insert = t[i]
        j = i
        while t[j - 1] > to_insert and j > 0:
            t[j] = t[j - 1]
            j -= 1
        t[j] = to_insert


def bubble_sort(t):
    sorted = False
    i = len(t) - 1
    while i > 0 or not sorted:
        sorted = True
        for j in range(i):
            if t[j] > t[j + 1]:
                swap(t, j, j + 1)
                sorted = False
        i -= 1

def counting_sort(t):
    v_max = max(t)
    count = [0] * (v_max + 1)
    for e in t:
        count[e] += 1
    output = [None for i in range(len(t))]
    i = 0  # indice de parcours du tableau résultat
    for v in range(v_max + 1):
        for j in range(count[v]):
            output[i] = v
            i += 1
    return output
```

```python
def complexity_counting_sort(t):
    v_max = max(t)
    count = [0] * (v_max + 1)
    for i in range(len(t)):
        count[t[i]] += 1
    for i in range(1, v_max + 1):
        count[i] += count[i - 1]
    output = [None for i in range(len(t))]
    for i in range(len(t)):
        output[count[t[i]] - 1] = t[i]
        count[t[i]] -= 1
    return output


def counting_sort_bis(t):
    if len(t) > 0:
        v_max = max(t)
        # v_max = t[0]
        # for i in range(1, len(t)):
        #     if t[i] > v_max:
        #         v_max = t[i]  # trouver le maximum
        count = [0 for _ in range(v_max + 1)]
        for i in range(len(t)):
            count[t[i]] += 1  # compter les occurrences de chaque t[i] de t.
            # Certaines cases de count sont nulles : pas de valeur
               correspondante dans t
        for i in range(1, v_max + 1):
            count[i] += count[i - 1]  # cumul des effectifs
        output = [None for _ in range(len(t))]
        for i in range(len(t)):
            count[t[i]] -= 1  # on compte à partir de 0
            output[count[t[i]]] = t[i]  # à la bonne place !
            # print("elem :", t[i], "place :", count[t[i]], output)
        return output
    else:
        return None


def sort_timing():
    sizes = [i for i in range(10, 5000, 500)]
    M = 10 * max(sizes)
    results = []
    for i in range(len(sizes)):
        t = [randint(0, M) for _ in range(sizes[i])]
        mem_t = t[:]
        results.append([])
        for method in [selection_sort, bubble_sort, insertion_sort,
                       counting_sort, sorted]:
            t = mem_t[:]
            tic = time.perf_counter()
            method(t)
            toc = time.perf_counter()
            results[i].append(toc - tic)
        # print("#", i, " - ", sizes[i], " -> ", results)
```

```
    sel = [results[i][0] for i in range(len(sizes))]
    bub = [results[i][1] for i in range(len(sizes))]
    ins = [results[i][2] for i in range(len(sizes))]
    cnt = [results[i][3] for i in range(len(sizes))]
    p = [results[i][4] for i in range(len(sizes))]

    from matplotlib import pyplot as plt

    plt.figure()
    plt.plot(sizes, sel, color='red', label='selection sort')
    plt.plot(sizes, bub, color='green', label='bubble sort')
    plt.plot(sizes, ins, color='blue', label='insertion sort')
    plt.plot(sizes, cnt, color='cyan', label='counting sort')
    plt.plot(sizes, p, color='black', label='sorted python')

    plt.xlabel('n', fontsize=18)
    plt.ylabel('time', fontsize=16)
    plt.legend()
    plt.show()


# MAIN PROGRAM
N = 20
M = 100

t = [randint(0, M) for _ in range(N)]

for method in [selection_sort, insertion_sort, bubble_sort, counting_sort]:
    tmp = t[:]
    if method in [selection_sort, bubble_sort, insertion_sort]:  # in-place
        method(tmp)
    else:
        tmp = method(tmp)  # counting_sort not in-place
    print("--> Method", method.__name__, ":\n ", t, "\n", tmp)

t = ["Zorglub", "Spirou", "Fantasio", "Marsupilami", "Marsu", "Samovar", "
    Zantafio"]
for method in [selection_sort, insertion_sort, bubble_sort]:  # , counting_sort
    ]:
    method(t)
    print("--> Method", method.__name__, ":\n ", t, "\n", t)

sort_timing()
```



B Recherche d'un élément dans un tableau

On considère une liste `L` contenant des éléments de type `int`. Cette liste est **triée** par ordre croissant de ses éléments. On veut savoir si un élément `x` est présent dans `L` et comparer les performances des approches séquentielles et la dichotomiques. On dispose des algorithmes 4, 5 et 6.

Algorithme 4 Recherche séquentielle d'un élément dans un tableau

Fonction RECHERCHE_SÉQUENTIELLE(t, elem)
  n ← taille(t)
  pour i de 0 à $n-1$ **répéter**
    si t[i]= elem **alors**
      renvoyer i ▷ élément trouvé, on renvoie sa position dans t
  renvoyer l'élément n'a pas été trouvé

B1. Coder l'algorithme de recherche séquentielle d'un élément dans un tableau. Lorsque l'élément n'est pas présent dans le tableau, la fonction Python renvoie `None`. Sinon, elle renvoie l'indice de l'élément trouvé dans le tableau. Vérifier que cet algorithme fonctionne sur un tableau d'entiers de 20

Algorithme 5 Recherche d'un élément par dichotomie dans un tableau trié

```
Fonction RECHERCHE_DICHOTOMIQUE(t, elem)
    n ← taille(t)
    g ← 0
    d ← n-1
    tant que g ⩽ d répéter            ▷ ⩽ cas où valeur au début, au milieu ou à la fin
        m ← (g+d)//2                  ▷ Division entière : un indice est un entier !
        si t[m] < elem alors
            g ← m + 1                 ▷ l'élément devrait se trouver dans t[m+1, d]
        sinon si t[m] > elem alors
            d ← m - 1                 ▷ l'élément devrait se trouver dans t[g, m-1]
        sinon
            renvoyer m                ▷ l'élément a été trouvé
    renvoyer l'élément n'a pas été trouvé
```

Algorithme 6 Recherche d'un élément par dichotomie dans un tableau trié, renvoyer l'indice minimal en cas d'occurrences multiples.

```
Fonction RECHERCHE_DICHOTOMIQUE_INDICE_MIN(t, elem)
    n ← taille(t)
    g ← 0
    d ← n-1
    tant que g < d répéter            ▷ attention au strictement inférieur !
        m ← (g+d)//2                  ▷ Un indice de tableau est un entier !
        si t[m] < elem alors
            g ← m + 1                 ▷ l'élément devrait se trouver dans t[m+1, d]
        sinon
            d ← m                     ▷ l'élément devrait se trouver dans t[g, m]
    si t[g] = elem alors
        renvoyer g
    sinon
        renvoyer l'élément n'a pas été trouvé
```

éléments rempli aléatoirement. Dans le pire des cas, quel est le coût d'une recherche séquentielle en fonction de la taille du tableau ?

B2. Coder l'algorithme 5. Lorsque l'élément n'est pas présent dans le tableau, la fonction Python renvoie `None`. Sinon, elle renvoie l'indice de l'élément trouvé dans le tableau. Vérifier que cet algorithme fonctionne sur un tableau d'entiers de 20 éléments rempli aléatoirement et trié.

B3. Coder l'algorithme 6. Vérifier que cet algorithme fonctionne sur un tableau d'entiers de 20 éléments rempli aléatoirement et trié et que l'indice renvoyé est bien l'indice minimal de la première occurrence de l'élément recherché.

B4. On suppose que la longueur de la liste est une puissance de 2, c'est à dire $n = 2^p$ avec $p \geqslant 1$. Combien d'étapes l'algorithme 6 comporte-t-il ? En déduire le nombre de comparaisons effectuées, dans le cas où l'élément est absent, en fonction de p puis de n, et comparer avec l'algorithme de recherche séquentielle.

Solution : Supposons que la taille de la liste soit une puissance de 2 : $n = 2^p$. Soit k, le nombre de tours de boucle nécessaires pour terminer l'algorithme. À la fin de l'algorithme, le tableau ne contient qu'un seul élément. Comme on divise par deux la taille du tableau à chaque tour de boucle, à la fin de l'algorithme on a nécessairement :

$$1 = \frac{n}{2^k} = \frac{2^p}{2^k}$$

On en déduit que $k = \log_2 n$. On dit que cet algorithme est de complexité logarithmique en $O(\log n)$ (cf. cours du deuxième semestre)

B5. Tracer le graphique des temps d'exécution des algorithmes précédent en fonction de n. Les tracés sont-ils cohérents avec les calculs des coûts effectués précédemment ?

B6. La recherche dichotomique fonctionne-t-elle sur les listes non triées ? Donner un contre-exemple si ce n'est pas le cas.

Solution :

Code 3 – Recherche un élément dans un tableau

```python
import time
from random import randint


def seq_search(t, elem):
    for i in range(len(t)):
        if t[i] == elem:
            return i
    return None


def dichotomic_search(t, elem):
    g = 0
    d = len(t) - 1
    while g <= d:
```

```python
        m = (d + g) // 2
        # print(g, m, d)
        if t[m] < elem:
            g = m + 1
        elif t[m] > elem:
            d = m - 1
        else:
            return m
    return None


def dichotomic_search_left_most(t, elem):
    g = 0
    d = len(t) - 1
    while g < d:
        m = (d + g) // 2
        # print(g, m, d)
        if t[m] < elem:
            g = m + 1
        else:
            d = m
    if t[g] == elem:
        return g
    else:
        return None


def search_timing():
    sizes = [i for i in range(10, 100000, 500)]
    M = 5 * max(sizes)
    results = []
    for i in range(len(sizes)):
        t = sorted([randint(0, M) for _ in range(sizes[i])])
        mem_t = t[:]
        results.append([])
        for method in [seq_search, dichotomic_search,
                       dichotomic_search_left_most]:
            t = mem_t[:]
            tic = time.perf_counter()
            method(t, M // 4)
            toc = time.perf_counter()
            results[i].append(toc - tic)
        print(f"#{i} - {sizes[i]} -> {results}")

    seq = [results[i][0] for i in range(len(sizes))]
    dics = [results[i][1] for i in range(len(sizes))]
    dicslm = [results[i][2] for i in range(len(sizes))]

    from matplotlib import pyplot as plt

    plt.figure()
    plt.plot(sizes, seq, color='cyan', label='Sequential search')
    plt.plot(sizes, dics, color='blue', label='Dichotomic search')
    plt.plot(sizes, dicslm, color='black', label='Dichotomic search left most')
    plt.xlabel('n', fontsize=18)
```

```
    plt.ylabel('time', fontsize=16)
    plt.legend()
    plt.show()

# MAIN PROGRAM
t = [0, 1, 2, 4, 7, 7, 9, 13, 17, 65, 99, 99, 99, 99, 101, 111, 111, 111, 113]

print(t)
for value in t:
    print(value, seq_search(t, value), dichotomic_search(t, value),
       dichotomic_search_left_most(t, value),
         dichotomic_search(t, value) == dichotomic_search_left_most(t, value))

search_timing()
```