Recherche et Tri

Le Gourrierec Alan

October 8, 2023

Unité d'enseignement : Algorithmique et programation impérative

Enseignant : P. Marteau

Établissement : ENSIBS (Vannes)

Contents

Introduction

Durant ce cours compte rendu, je vais vous présenter les diverses activités que nous avons réalisé au cours de ce TP2 en programmation impérative qui porte majoritairement sur les algorithmes de tri et la recherche dans une liste, tirer ou non.

Ces problème sont des problème récurant dans le monde contemporain avec des recherches pour pouvoirs résoudre ses problèmes le plus vite possible. Nous verrons donc, dans le cadre du tri d'une liste 3 algorithmes majeurs :

- Le tri par sélection : il consiste à regarder une, par exemple de nombre, et dès le trier par ordre croissant en parcourant la liste entièrement pour trouver le plus faible.
- le tri par insertion : il consiste à ranger la valeur à l'emplacement où elle devrait être en utilisant une liste secondaire, par exemple.
- le tri fusion : il utilise le fais de scinder la liste en deux listes de façon récursive en effectuant un tri sur chacune de liste, puis à la fin en les rajeunissants.

Chapter 1

Algorithme de recherche

1.1 Recherche dans une liste non trier

L'algorithme de recherche qui me paraissait le plus logique à faire est l'algorithme de recherche où nous parcourons toute la liste, car cette dernière n'est pas triée. Il nous faudra donc, dans le pire des cas n itération (n représentant pas la taille du tableau). J'ai donc réalisé l'algorithme de la manière suivante : Je souhaite que mon programme compare toutes les chaînes de caractère

du tableau avec lui-même, si cette chaîne de caractère est la même alors je renvoie sa position dans la liste.
Si ce mot n'existe pas alors je parcours toute la liste et je renvoie -1

```
def Recherche(mot,liste):
    i=0
    try:
        while mot != liste[i]:
            i+=1
        return i+1
    except:
        return "désoler, vôtre mot n'existe pas dans la liste"

mot = "camion"
T=["test","baltimore","camion","oud-point","abri-bus","cailloux"]
print(Recherche(mot,T))
```

Figure 1.1: Recherche

J'ai donc testé cet algorithme avec le mot "camion" et il prend donc 3 itérations avant d'arriver à la fin. Il en va de même pour "test" qui ne demandera pas qu'une unique comparaison et pour "cailloux" qui lui seras le plus long à atteindre.Pour prendre en compte toutes les possibilités, il faut aussi prendre en compte le cas où le mot n'existerait pas, prenons l'exemple de "les points virgules me manque".

```python
def Recherche(mot,liste):
    i=0
    try:
        while mot != liste[i]:
            i+=1
        return i+1
    except:
        return "désoler, vôtre mot n'existe pas dans la liste"

mot = "les points virgules me manque "
T=["test","baltimore","camion","oud-point","abri-bus","cailloux"]
print(Recherche(mot,T))
```

Figure 1.2: rechercher avec un mot non présent dans la liste

Dans ce cas, le code va nous renvoyer : "désolé, votre mot n'existe pas dans la liste" (et non pas -1 comme demandé dans la consigne) contrairement à l'emplacement de ce mot dans la liste pour les autres.

1.2 Recherche dans une liste triée

Ce cas est bien différent du précédent, dans ce cas, plusieurs possibilités s'offrent à nous, comme par exemple, la solution de la recherche diatomique. Cette recherche est celle que nous allons utiliser lorsque nous utilisons un dictionnaire par exemple (le code se trouve dans le zip que je vous ai fourni.).

Pour réaliser cette recherche, il faut que dans un premier cas, nous regardions si le mot du milieu de cette liste est supérieur ou inférieur à notre mot. S'il est supérieur, alors nous prendrons la moitié supérieure de la liste, ou nous prenons la moitié inférieure.
Nous répéterons ceci tant que nous n'avons pas trouvé le mot que nous recherchons.

Nous obtenons l'algorithme en python suivant :

```python
import time

PATH = 'programation-imperative/TP2/pyramide/recherche/Dictionaire.txt'
f = open(PATH, 'r')
dico = f.read().split("\n")
f.close

def Dicothomie(mot,dico):
    placem,placeM,t = 0,len(dico),time.time() #initialisation de l'encadrement de la liste
    place = (placem+placeM)//2 #définition de place qui désigne la position que la liste va pointé
    s,splus1 = None,dico[place] #définition de deux string qui permetrons de savoir si un mot exite ou non

    while(mot != dico[place]):
        if(mot>dico[place]):
            placem = place

        elif(mot<dico[place]):
            placeM = place

        if(s == splus1):
            return "le mot n'existe pas"

        place=(placem + placeM)//2
        s = splus1
        splus1 = dico[place]

    return place, time.time() -t

print(Dicothomie("toto",dico))
```

Figure 1.3: code python de la dicotomie

Suite à divers test (en prenant un mot se trouvant à la fin, au milieu ou au début) nous obtenons toujours le même nombre d'itération : 16. Ceci est sensiblement la même chose pour les mots qui n'existe pas et nous obtenons, un message nous disant que ce mot n'existe pas.

1.3 Recherche dans une chaîne de caractère

Pour cet algorithme, nous souhaitons rechercher la présence d'un mot dans une chaîne de caractère :

Pour ce faire, nous allons découper nos chaînes de caractères (le mot et la phrase) en tableau de char. Si le premier caractère de mot correspond à l'un de ceux de la phrase, alors nous regardons le suivant jusqu'à que l'algorithme trouve (et renverras la position où se trouve le premier caractère) sinon ne renverras rien.
Il faut aussi prendre en compte le cas où le mot est vide ("" en python) ou la phrase, dans ce cas-là, nous renverrons une erreur.

En python, cette algorithme ressemble à ceci :

```python
def RechercheChaine(mot,liste,start):
    T,L=[],[]
    T=list(mot)
    L=list(liste)
    if len(T) == 0 or len(L) == 0:
        return "longueure des arguments insufisants"
    for i in range(start,len(L)):
        a = 0

        while T[a] == L[i+a]:
            if T[a] == L[i+a] and a == len(T)-1:
                return i
            a+=1
    return None
```

Figure 1.4: Recherche dans une chaine de caractères

Le nombre d'itération dépend d'où se trouve la chaîne de caractère, de quelle est la taille de la phrase et du mot. Dans le pire des cas, la complexité de notre algorithme est n.

1.4 Remplacement

Ce dernier algorithme a pour but de remplacer une chaîne de caractère par une autre. Pour se faire, nous allons procéder de la manière suivante :

Nous allons réutiliser l'algorithme Recherche d'une chaîne de caractère (**??**) pour obtenir la position de chacune des itérations. Pour chaque itération, nous allons couper la liste en 2 partie (gauche et droite), puis placer entre les deux une nouvelle chaîne de caractères, et ce, jusqu'à la fin. Si le mot ou la phrase est de longueur 0, alors nous revoyons une erreur sinon nous revoyons la phrase modifier.

```
def Remplacement(mot,phrase,remplacer):
    i = 0
    if len(mot) ==0 or len(phrase)==0 :
        return "longueure des arguments insufisante"

    while i != None:
        gauche,droite = [],[]

        i = RechercheChaine(mot,phrase,i)
        if i == None:
            return phrase

        else :
            for j in range(i):
                gauche.append(phrase[j])

            for j in range(i+len(mot),len(phrase)):
                droite.append(phrase[j])

        phrase = "".join(gauche)+remplacer+"".join(droite)
```

Figure 1.5: Algorithme de remplacement

Même complexité que le précédent au fait près que nous allons effectuer cette opération plusieurs fois.

Chapter 2

Algorithme de tri

2.1 tri par sélection

Ceci est l'algorithme le plus simple au quelle nous pouvons penser mais aussi le plus lent. Cette algorithme va regarder toutes les valeurs dans les valeurs dans la liste les trier celons ce qui est dermander. Prenons l'exemple du tri par ordre croissant :
Il va faloir, dans le pire cas (le premier) parcourir toute la liste et comparer la valeur la plus faible trouvé avec la nouvelle valeur du tableau puis enfin échanger la valeur la plus faible avec celle qui occupe la permière position

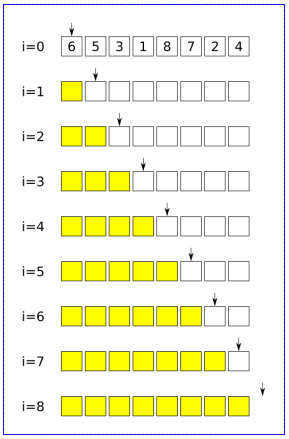

Figure 2.1: tri par selection

2.2 Tri par insertion

Ce second algorithme est le tri par insertion. Il consiste simplement en la création d'une seconde liste ou nous placerons les valeurs directement à l'emplacement ou elles doivent aller. Par exemple, nous prenons la première lettre de notre tableau non trier, par exemple j, nous la rangeons donc à la position 10 et nous répétons l'opération jusqu'a que toute la liste sois trié. Il y à donc deux opération pour chaque objet trié : la convertion vers une valeur numérique (dans notre cas car nous trions les lettres de l'alphabet) et le rangement dans la liste secondaire. La complexité est donc de O(2n)

2.3 trifusion

Cette algorithme est le plus rapide des 3, il consiste en un tri en utilisant la récurtion. Pour se faire, nous allons diviser la liste en deux et la trier en même temps et nous allons répéter l'action jusqu'à que la liste ne sois composé que d'un élément, après ceci, nous allons racelbler ces élément dans la liste principale. La complexité de cette algorithme est de $O(nlog(n))$

Conclusion

Durant ce TP, nous avons donc pu comprendre, aprendre à réaliser des algorithme de rechercher et de tri. La dicotomie est celui qui m'a particulièrement surpis, il est d'une vitesse incroyable mais je me questionne aussi sur le fais de svoir si il est le meilleur pour des liste courte.
L'algorithme de tri fusion m'a été plutôt compliquer à réaliser par manque d'habitude vis à vis de la récursivité. J'ai donc pu comprendre plus en profondeur comment coder de manière récursive et aussi plus en profondeur un langague que je ne maitrise pas à la perféction.