Complexité des algorithmes

Compétences

- Prévoir la complexité théorique d'un algorithme.
- Étudier la complexité temporelle expérimentale d'un algorithme

Optimisation des espaces dans un texte

On considère le problème de la composition équilibrée d'un paragraphe dans un traitement de texte. Le texte d'entrée est une séquence de n mots de longueurs $l_1, l_2, \ldots, l_n$ mesurées en nombre de caractères. On souhaite organiser les mots de ce paragraphe de manière équilibrée sur un nombre défini de ligne contenant chacune exactement N caractères. Chaque ligne doit comporter un certain nombre de mots tous séparés par une espace et des caractères d'espacement supplémentaires complètent la ligne pour qu'elle contienne exactement N caractères. Un mot ne peut pas être coupé.
Si une ligne de rang ($p \in \mathbb{N}$) contient les mots de i à j, le nombre de caractères d'espacement supplémentaires ($c \in \mathbb{N}$) nécessaires pour compléter la ligne est égal au nombre de caractères de la ligne N, moins le nombre de caractères nécessaires pour écrire les mots moins le nombre de caractères nécessaires pour séparer les mots ($j-i$).

$$c_p = N - (j-i) - \sum_{k=i}^{j} l_k$$

Pour un paragraphe équilibré le critère d'équilibre retenu est le suivant : On souhaite minimiser la somme, sur toutes les lignes hormis la dernière, des cubes des nombres de caractères d'espacement supplémentaires de fin de ligne. Soit m le nombre de lignes nécessaires pour un texte équilibré nous avons :

$$c = \sum_{\text{ligne}=1}^{m-1} \left(N - \sum_{k=i}^{j} l_k - (j-i) \right)^3_{\text{ligne}} \quad \text{qui doit être minimum}$$

1. La solution naïve est gloutonne : on place le maximum de mots sur la première ligne, puis on recommence pour la deuxième ligne et ainsi de suite tant qu'il reste des mots. Écrire une fonction itérative `compositionGlouton(text, n, N)` qui prend en paramètre un tableau de mots qui représente le texte, le nombre d'éléments dans le tableau, le nombre maximum de caractère dans une ligne et qui renvoie la somme des cubes des espaces supplémentaires sur toutes les lignes exceptée la dernière.
   Exemple : avec une ligne de 20 caractères

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
E	n		o	u	v	r	a	n	t		u	n		v	i	e	u	x	
c	o	f	f	r	e	t		v	e	n	u		d	e					
l	'	o	r	i	e	n	t												

   La première ligne contient 1 espace supplémentaire et la deuxième 5 espaces supplémentaires pour un coût de

$$c = \sum_{\text{ligne}=1}^{2} c_{\text{ligne}}^3 = 1^3 + 5^3 = 126$$

2. Quelle est la complexité de votre algorithme?
3. Une solution optimale est donnée par l'algorithme suivant :

```
1  def composition(text, n, N):
2      c = [0]*(n+1)
3      for i in range(n-1, -1, -1):
4          if n-i + sommeCaracteresMots(text, n, i) <= N:
5              c[i] = 0
6          else:
7              c[i] = 1e12
8              for j in range(i, n):
9                  if (j - i) + sommeCaracteresMots(text, j+1, i) <= N:
10                     if (N-j+i-sommeCaracteresMots(text,j+1,i))**3+c[j+1]<c[i]:
11                         c[i]=(N-j+i-sommeCaracteresMots(text,j+1,i))**3+c[j+1]
12
13     return c[0]
```

```python
def sommeCaracteresMots(text, n, i):
    if i < n+1:
        somme = 0
        for i in range(i, n):
            somme += len(text[i])
        return somme
    return -1
```

En utilisant la phrase de la question 1, comparer les résultats donnés par votre algorithme glouton et la fonction `composition` pour des lignes contenant entre 20 et 24 caractères inclus. Normalement les deux fonctions doivent donner les mêmes résultats. Pour vous en assurer, vous pouvez utiliser le script suivant :

```python
text = "En ouvrant un vieux coffret venu de l'Orient"
text = text.split(" ")
for i in range(20, 25):
    print("i = ", i)
    print("Glouton :", compositionGlouton(text, len(text), i))
    print("Dynamique :", composition(text, len(text), i))
    print()
```

4. Recommencer avec la phrase suivante : `Homme libre, toujours tu chériras la mer indomptable et sauvage.` Que constatez vous ?
5. Quelle est la complexité de la fonction `composition` ?
6. **Question Optionelle :** Écrire une fonction récursive optimale `compositionRecursive(text, i, n, N)` qui prend en paramètres un texte sous la forme d'un tableau de mots, un indice de départ pour la lecture du texte, le nombre de mots, la longueur d'une ligne et qui renvoie c. Cette fonction teste toutes les possibiltés de composition pour ne garder que celle qui répond au critère d'équilibre. La condition d'arrêt de cette fonction est la suivante :

```python
if n - i + sommeCaracteresMots(text, n, i) <= N:
    return 0
```

Quelle est la complexité de cette fonction récursive ?

7. Écrire un script python permettant de comparer la complexité temporelle expérimentale des différents algorithmes avec le texte : **Elle cria je prends, je rachète tout ça, ce que vous vendez là, c'est mon passé à moi."** .

Pour manipuler le temps avec Python vous disposez du module `time`.

```python
import time
...
start = time.perf_counter()
# Ici le code dont vous voulez mesurer
# le temps d'exécution
interval = time.perf_counter() - start
```

Complexité en image

Notions de base sur les images Python

Les bouts de code suivants pourront être testés avec l'image `lion.jpeg` contenue dans le dossier `TD_machine/images` du github.

Lire une image

```python
from PIL import Image

img = Image.open("path/mon_image.png")
print(img.format, img.size, img.mode)
img.show()
```

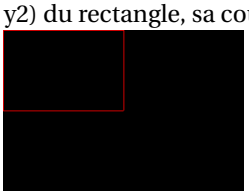
Fabriquer une image

```
# création d'une image de même taille
#que mon_image.png
img1 = Image.new("RGB", img.size)
img1.show()
```

Changer le contenu d'un pixel avec `putpixel`

```
YELLOW = (255, 255, 0)
img1 = Image.new("RGB", (300, 200))
j = 100
for i in range(img1.size[1]):
    img1.putpixel((i, j), YELLOW)
img1.show()
```

1. Que remarquez-vous ?
2. Modifiez le code pour que la ligne jaune traverse la fenêtre.
3. Dessiner un les contours d'un rectangle rouge dont le coin supérieur gauche est en (0, 0) et le coin inférieur droit en (150, 100)
4. Modifier votre code pour que le coin inférieur droit soit systématiquement au milieu de la fenêtre quelque soit sa taille.
5. Écrire une fonction :
   `solidRectangle(img, x1, x2, y1, 2, color)`
   qui prend en paramètres une image, la position du coin supérieur gauche (x1, y1) et du coin inférieur droit (x2, y2) du rectangle, sa couleur et qui dessine un rectangle plein.




Récupérer le contenu d'un pixel avec `getpixel`

```
pixel = img.getpixel((i, j))
print(pixel)
```

Pour s'exercer

1. créer une image "RGB" aux dimensions (300, 300)
2. Écrire une fonction `stripesColoured(image, n)` qui divise l'image en `n` bandes verticales de couleurs différentes. Pour cela il sera judicieux de s'aider de la fonction `solidRectangle`

Niveaux de couleurs et complexité

Un pixel possède une valeur qui peut-être un scalaire (un nombre) et représenter un niveau de gris, ou un vecteur souvent à 3 composantes représentant chacune une couleur ayant des niveaux d'intensité $[0, 255]$.

Pour refaire les niveaux d'une image, on peut utiliser la méthode suivante :

- pour chaque couleur, on cherche la valeur minimum v_{min} et la valeur maximum v_{max} apparaissant parmi les pixels de l'image, on obtient la gamme de cette couleur : l'intervalle $[v_{min}, v_{max}]$
- puis, on réaffecte à chaque pixel une nouvelle valeur v_N sur cette couleur, calculée en projetant l'intervalle de départ sur l'ensemble des valeurs disponibles $[0,255]$. On obtient cette valeur à partir de la valeur initiale v_i de la couleur correspondante dans le pixel à modifier, en calculant :

$$v_N = \frac{255(v_i - v_{min})}{v_{max} - v_{min}}$$

1. Écrivez une fonction `minimumR(img)` qui calcule la valeur minimale des coefficients de rouge sur toute l'image. Écrivez de même une fonction `maximumR(img)`.
2. Modifiez les fonctions pour qu'elles renvoient les minimum et maximum de chaque couleur.
3. Écrivez le programme qui refait les niveaux de couleurs de l'image avec la méthode précédente. Vous pourrez tester votre programme en déchiffrant l'image `hell.png`.
4. Quelle est la complexité de votre algorithme?

ModifiedBinarySearch

Introduction

L'algorithme modifié de dichotomie est un algorithme présenté par [**?**] dans un article de recherche. Cet algorithme optimise le cas le plus défavorable de l'algorithme de dichotomie en comparant l'élément recherché avec le premier et le dernier élément du tableau, ainsi que l'élément central. Il vérifie également que l'élément recherché, un nombre dans notre cas, appartient à la plage de nombres présents dans le tableau à chaque itération, ce qui permet de réduire le temps pris par les pires cas de l'algorithme de recherche dichotomique. Dans ce TP nous allons essayer de suivre les différentes étapes de cet article pour mieux comprendre les améliorations approtées par cet algorithme.

Concept of Modified Binary Search

Le concept de recherche dichotomique utilise uniquement l'élément du milieu pour vérifier s'il correspond à l'élément recherché noté x. Pour cette raison, si x est présent à la première position, la recherche prend plus de temps, ce qui en fait le pire cas de l'algorithme.

Pour améliorer les performances de la recherche dichotomique, l'algorithme modifié de la recherche dichotomique ne vérifie pas seulement si x est présent pour l'index du milieu mais il vérifie également s'il est présent à la première et dernière position du tableau intermédiaire pour chaque itération. Cet algorithme optimise aussi le cas où x n'est pas dans le tableau et distingue dans cette situation deux cas, celui où la valeur de x est en dehors de la plage des valeurs du tableau et celui où la valeur de x est comprise entre le plus petit et le plus grand élément du tableau.

Algorithms

Binary Search

Écrire la fonction de recherche dichotomique `binarySearch(x, tab, n)` qui prend en arguments la valeur recherchée, un tableau d'entiers et le nombre de valeurs présentes dans le tableau. Cette fonction renvoie la position dans le tableau de la valeur recherchée. Si cette valeur ne se trouve pas le tableau la fonction renvoie -1.

Modified Binary Search

À partir des informations données dans écrire un fonction `modifiedBinarySearch(x, tab, n)` de l'algorithme modifié de recherche dichotomique. Cette fonction prend les mêmes arguments et renvoie les mêmes informations que la fonction `binarySearch`

Performance Analysis

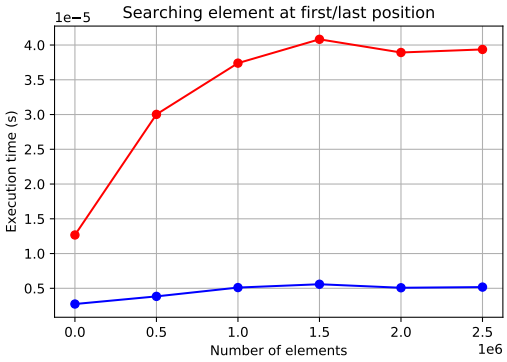
Soient $x \in \mathbb{N}$ la valeur recherchée et `tab[]` un tableau de valeurs entières. Dans cette partie il s'agit d'étudier le nombre de coups nécessaires pour atteindre x avec l'algorithme classique de dichotomie et l'algorithme modifié. Pour notre étude nous travaillerons avec le tableau d'entiers suivant :

`tab[]`

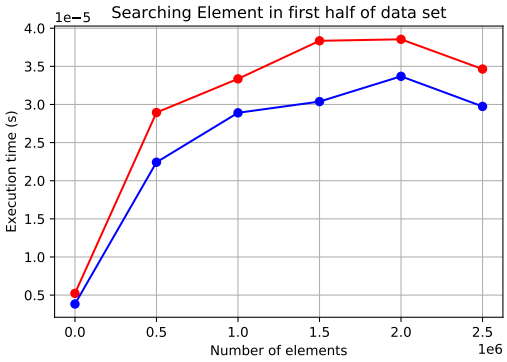
1	4	21	53	64	98	102	171	188	205
0	1	2	3	4	5	6	7	8	9

1. Appliquez les deux algorithmes à une valeur du tableau et vérifiez que l'index renvoyé est le bon.

2. Faire de même avec une valeur entière qui n'est pas dans le tableau.
3. Écrire un script python permettant de comparer la complexité temporelle expérimentale des deux algorithmes suivant la place de l'élément recherché (Fig1 et Fig2). Pour manipuler le temps avec Python disposez du module `time`.

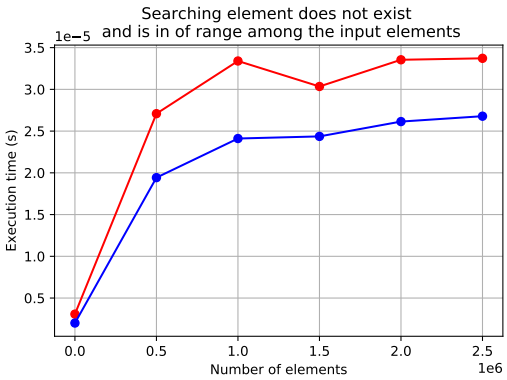


(a) First element

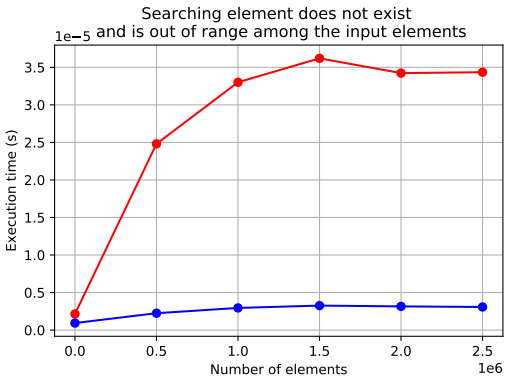


(b) First half

FIGURE 1 – Analyse des performances de l'algorithme de dichotomie modifié quand l'élément x occupe la première place ou une position quelconque dans la première moitié du tableau



(a) Element not in the array but in range of data set



(b) Element not in the array and out of range of data set

FIGURE 2 – Analyse des performances de l'algorithme de dichotomie modifié quand l'élément x n'est pas dans le tableau