TP3 MPSI ITC 2025 Tris, Recherche

À propos de la complexité

La *complexité temporelle* des algorithmes est une mesure de leur temps d'exécution. Comme les machines sur lesquels peuvent s'exécuter les codes ont des vitesses de processeurs et d'accès mémoire différentes, on adopte une mesure universelle : le nombre d'opérations en $O(1)$ (c.a.d à coût constant) effectuées pendant le déroulement du programme. Cette mesure est donnée en fonction de la *taille* de la ou des données d'entrées. Pour une liste de taille n passée en entrée, par exemple, on donne l'expression de la complexité en fonction de n.

Par ailleurs, on ne cherche pas à effectuer un décompte précis de la complexité, seulement une valeur simple à exprimer majorant cette complexité à « coefficient multiplicatif près ».

Il y a 3 types de complexité temporelle :

— La *complexité au mieux* : elle indique la plus courte durée d'exécution (en nombre d'opérations) pour une (ou des) données de taille fixe.
Par exemple, la complexité pour trier une liste de taille n avec certains algorithmes est plus petite si la liste est ordonnée dans le sens voulu.

— La *complexité au pire* : elle indique la pire durée d'exécution (en nombre d'opérations) pour une (ou des) données de taille fixe.
Reprenant l'exemple des tris, certains algorithmes s'exécutent plus lentement avec un liste triée dans l'ordre inverse de celui voulu.

— La *complexité en moyenne* nécessite de connaître des informations statistiques sur les données d'entrées afin d'être calculée. Par exemple, on peut supposer que les différentes listes passées en paramètre d'un algorithme de tri ont toutes la même probabilité d'apparaître (distribution uniforme). Parfois ausi, on peut supposer que les listes dont les éléments ont un ordre totalement aléatoire ont peu de chance d'être rencontrées et que le plus souvent, des listes « presque triée » sont passées en paramètres.

La *complexité en mémoire* des algorithmes s'intéresse à la quantité de mémoire nécessaire (en dehors des données d'entrées) pour leur exécution. Les catégories sont les mêmes que celles décrites ci-dessus.

Dans ce TP, on s'intéresse aux complexités temporelles au pire et au mieux.

1 Exercices

Question 1. Tri à bulle

Objectif : Écrire une fonction qui trie un tableau en utilisant l'algorithme du tri à bulle.

Description : Le tri à bulle, ou *bubble sort*, est un algorithme de tri simple qui fonctionne de la manière suivante :

— Le tableau est parcouru en comparant successivement chaque paire d'éléments adjacents.
— Si deux éléments sont dans le mauvais ordre (pour un tri croissant, si l'élément de gauche est supérieur à l'élément de droite), ils sont échangés.
— Ce processus est répété pour l'ensemble du tableau plusieurs fois. À chaque passage, l'élément le plus grand "bulle" vers la fin du tableau et se retrouve à sa position finale.
— Lorsque plus aucun échange n'est nécessaire, le tableau est considéré comme trié.

1. Écrire une fonction `passe(t:list,i:int)->bool` qui prend en paramètre un tableau et un indice de fin de parcours et implante une passe du tri à bulle. Le tableau est modifié durant le déroulement d ela fonction.

L'algorithme fait un parcours du tableau jusqu'à la position i (exclue) passée en paramètre et réalise un échange entre la valeur de la position courante et la suivante si l'ordre n'est pas bon.
Si un échange a eu lieu pendant une passe, la fonction renvoie `True` et `False` sinon.

```
t=[10,3,2,15,12,13,25,26]
res=passe(t,len(t))
t,res,passe([10,20,30],3)
```

```
([3, 2, 10, 12, 13, 15, 25, 26], True, False)
```

Identifier un meilleur cas et un pire cas pour la complexité de l'appel `passe(t,i)`.

2. Écrire une fonction `bulle(t:list)->None` qui réalise des passes jusqu'à ce qu'aucun échange n'ait lieu. La fonction de passe est appelée successivement avec des indices décroissants.
3. Etudier la complexité de l'appel `passe(t)` pour une liste de taille n.

```
t=[10,3,2,15,12,13,25,26]
bulle(t)
t
```

```
[2, 3, 10, 12, 13, 15, 25, 26]
```

Avant d'attaquer un nouvel exercice, faisons les rappels suivants :

```
t=[20,10,3,8,9,12,6]
print(t[2:4])
print(t[:4])
print(t[3:])
t.extend([100,200])
print(t)
```

```
[3, 8]
[20, 10, 3, 8]
[8, 9, 12, 6]
[20, 10, 3, 8, 9, 12, 6, 100, 200]
```

Question 2. Tri insertion

On s'intéresse au *tri insertion* : un tri *en place* (pas d'utilisation de tableau auxiliaire) et *stable* (pas de modification de l'ordre des *items* égaux).

1. Écrire une fonction `placer(liste:list, i:int)->None` qui place `liste[i]` à sa bonne position dans `liste[:i+1]`, en supposant que `liste[:i]` est déjà triée.

```
t = [10,15,20,25,12,60,23]
placer(t,4)
print(t)
```

```
[10, 12, 15, 20, 25, 60, 23]
```

La liste passée en paramètre subit des *effets de bords*.

2. Le *tri insertion* applique itérativement la fonction de placement. Écrire la fonction `tri_insertion(liste:list)->None` qui trie *en place* une liste.
3. Étudier la complexité de ce tri.
4. Qu'est-ce qui garanti le caractère *stable* de ce tri dans votre code ?

Question 3. Tri fusion

On donne le principe du *tri fusion* :
— On coupe en deux parties à peu près égales les données à trier ;
— On trie les données de chaque partie (pour cela, on coupe chaque partie en deux et on trie chacune) ;
— On fusionne les deux parties.
Le cas d'arrêt de cet algorithme récursif intervient lorsque la liste possède zéro ou un élément : elle est alors triée.

1. Écrire la fonction `fusion(t1:list,t2:list)->list` qui prend en paramètre deux listes ordonnées dans le sens croissant et retourne une troisième liste ordonnée qui contient tous les éléments des deux autres et aucun nouvel élément.
   Aucune des deux listes passées en paramètre n'est modifiée.

```
t1= [10,20,30]
t2 = [5,8,10,12,22,33]
fusion(t1,t2)
```

```
[5, 8, 10, 10, 12, 20, 22, 30, 33]
```

2. Écrire la fonction `tri_fusion(t: list) -> list` qui trie une liste selon la méthode du tri fusion. La liste passée en paramètre n'est pas modifiée.
3. Établir une relation de récurrence suivie par la complexité de l'appel `tri_fusion(t)` en fonctions de n, la taille de t.
4. Exprimer la complexité de `tri_fusion(t)` en fonction de n.
   Indication. On supposera d'abord que n est une puissance de 2, puis on passera au cas général.

Question 4.

Recherche naïve dans une liste
Objectif : Écrire une fonction qui recherche linéairement un élément x dans une liste.
Description : On parcourt la liste de gauche à droite et on compare chaque élément à x. On peut soit renvoyer l'indice de la première occurrence, soit `-1` si x n'apparaît pas.

1. 'Ecrire une fonction `recherche_naive(t:list, x:any)->int` qui renvoie l'indice de la première occurrence de `x` dans `t`, ou `-1` si `x` n'est pas présent.

```
t = [10, 3, 2, 15, 12, 13, 25, 26]
recherche_naive(t, 15), recherche_naive(t, 7)
```

```
(3, -1)
```

2. Identifier le *meilleur cas*, le *pire cas* pour la complexité en fonction de $n = |t|$. Justifier brièvement.

Question 5.

Recherche dichotomique (binaire)
Objectif : Écrire une fonction qui recherche un élément x dans une liste triée en temps avantageux.

Description : La liste *doit être triée en ordre croissant*. À chaque étape, on compare x avec l'élément du milieu et on élimine la moitié qui ne peut pas contenir x.

1. 'Ecrire une fonction `recherche_dicho(t:list, x:any)->int` qui renvoie l'indice de x s'il est présent, ou `-1` sinon. On supposera que les éléments sont comparables avec `<, ==, >`.

```
t = [2, 3, 10, 12, 13, 15, 25, 26]
recherche_dicho(t, 15), recherche_dicho(t, 7)
```

```
(5, -1)
```

2. Donner et justifier la complexité dans le meilleur cas, le pire cas (en fonction de n).
3. Comparer expérimentalement le nombre de comparaisons entre la recherche naïve et la recherche dichotomique pour des listes triées de grande taille (optionnel).

Question 6.

Recherche naïve d'un mot dans une chaîne

Objectif : Écrire une fonction qui recherche la première occurrence d'un motif (chaîne) dans un texte par la méthode naïve.

Description : On aligne le motif `mot` au début du `texte` et on vérifie caractère par caractère. En cas d'échec, on décale d'un caractère vers la droite et on recommence, jusqu'à trouver une occurrence ou atteindre la fin.

1. Écrire une fonction `start(t: str, m: str, i: int) -> bool` qui renvoie `True` si gboxm apparaît dans `t` à partir de la position `i`.

```
1 texte = "abracadabra"
2 start(texte,"ac",3), start(texte,"ac",0)
```

```
(True, False)
```

2. 'Ecrire une fonction `cherche_mot_naif(texte:str, mot:str)->int` qui renvoie l'indice de début de la première occurrence de `mot` dans `texte`, ou `-1` si le motif n'apparaît pas.

```
1 texte = "abracadabra"
2 cherche_mot_naif(texte, "cada"), cherche_mot_naif(texte, "cadabra"),\
3 cherche_mot_naif(texte, "xyz")
```

```
(4, 4, -1)
```

3. Étudier la complexité en fonction de $n = |\text{texte}|$ et $m = |\text{mot}|$: pire cas, meilleur cas, cas moyen (hypothèses à préciser).
4. Modifier la fonction pour renvoyer la liste de *toutes* les positions d'occurrence (si le motif peut apparaître plusieurs fois).