TD d'Algorithmique et structures de données
Tri fusion

Équipe pédagogique Algo SD

On considère le code du tri fusion suivant :

```
def triFusion(tableau):
    if len(tableau) == 1:
            return tableau
    else:
        milieu = len(tableau) // 2
        gauche = triFusion(tableau[:milieu])
        droite = triFusion(tableau[milieu:])
        return fusion(gauche, droite)
```

On supposera disposer d'une fonction de *fusion*, fusionnant deux tableaux triés en $O(n)$ comparaisons.

1 Analyse de coût

Afin de mieux cerner l'exécution du programme, on se propose de dessiner les différentes opérations réalisées sous forme de graphe. La figure 1 illustre les opérations de découpe et de fusion sur un tableau contenant les entiers de 1 à 4 (pour un tri par ordre décroissant).

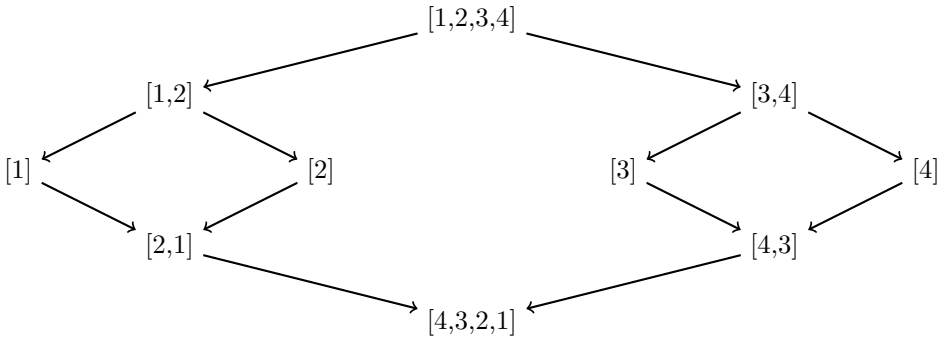

FIGURE 1 – Graphe de dépendances

1. Dessinez un graphe pour une taille n de tableau égale à 8.
2. Pour un tableau de taille n, combien de niveaux voit-on sur le graphe ?
3. Où ont lieu les opérations de fusion ? Annotez le graphe en rajoutant le coût de chaque fusion. En sommant les coûts sur chaque niveau donnez les coûts au pire cas, au meilleur cas, en moyenne du tri fusion.
4. Numérotez toutes les opérations de fusion en fonction de leur ordre d'exécution.
5. Quel est le coût en mémoire (précis) du tri ?

2 Version itérative

1. On se propose maintenant d'éliminer les appels récursifs tout en gardant une consommation mémoire faible. En changeant l'ordre de réalisation des fusions (en travaillant niveau par niveau) écrire un

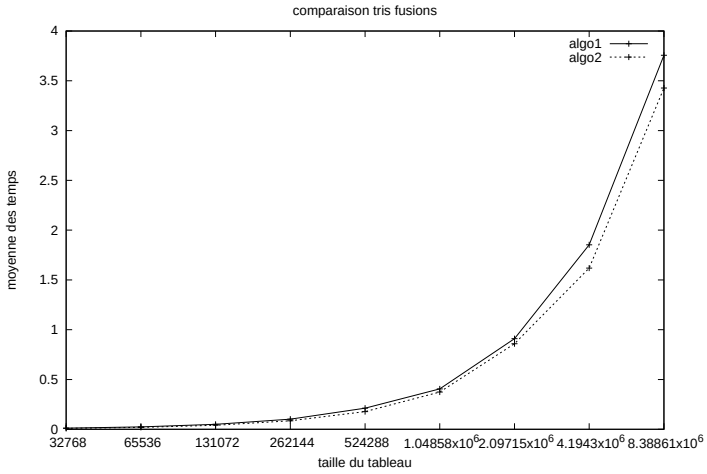

FIGURE 2 – temps d'exécution pour les deux algorithmes

algorithme itératif de tri fusion. Pour faire simple on pourra supposer que n est une puissance de 2.

2. On considère la courbe des temps d'exécutions obtenus en exécutant nos deux algorithmes. Pouvez-vous deviner quelle courbe est quel algorithme ?
3. Proposez une version récursive en profondeur d'abord jusqu'à une taille limite de 500Ko, puis terminant par la version itérative en largeur.

3 Combinaison avec le tri par insertion

Le tri par insertion est en pire cas plus mauvais que le tri fusion, puisqu'il est en $\Theta(n^2)$. Néanmoins, les constantes cachées dans le $\Theta()$ rendent en pratique cet algorithme plus rapide que le tri fusion pour de petites valeurs de n. On va donc combiner les deux, et remplacer les appels récursifs `triFusion(tableau[:milieu])` et `triFusion(tableau[milieu:])` par des tris par insertion dès que les sous-tableaux deviennent de taille k suffisamment petite. Le but de cette partie est de déterminer k.

1. Ecrire l'algorithme ainsi modifié. Montrer que n/k sous-tableaux, tous de taille k, peuvent être triés par insertion en $\Theta(nk)$ en pire cas.
2. Montrer que les n/k sous-tableaux triés peuvent être fusionnés en $\Theta(n \log(n/k))$ en pire cas.
3. Le coût de l'algorithme modifié est donc en $\Theta(nk + n \log(n/k))$ en pire cas, avec des constantes cachées différentes du tri fusion classique. En raisonnant uniquement de manière asymptotique en notation $\Theta()$, quelle est la valeur maximale de k, en fonction de n, pour laquelle cet algorithme modifié est au moins aussi performant que l'algorithme classique en pire cas ?