II. Tableaux et algorithmes de tris

1. Introduction aux tableaux

Suppsons que nous souhaitons utiliser avec un ordinateur, un tableau d'objets de type `T`. Si ce tableau est de taille $n \in \mathbb{N}$, il sera représenté en mémoire par n cellules consécutives contenant chacune un objet de type `T`. L'indice de la première cellule sera 0 tandis que celle de la dernière sera $n-1$. Voici par exemple un tableau de type chaîne de caractère et de taille 6 contenant des noms de mathématiciens.

0	1	2	3	4	5
Euler	Gödel	Artin	Noether	Cantor	Jones

Pour un tableau tab de type `T` et de longueur N nous disposons des primitives suivantes :

– $tab[i] \leftarrow obj$, avec $i \in [0, N-1]$ et obj un objet de type `T`, écris obj dans la case d'indice i du tableau tab ;

– $obj \leftarrow tab[i]$, avec $i \in [0, N-1]$ et obj un objet de type `T`, écris le contenu de la case d'indice i de tab dans la variable obj ;

– longueur(tab) qui retourne la longueur du tableau tab.

Exercice. Ecrire les fonctions suivantes :

1. EST_POSITIF(tab) qui étant donné un tableau d'entiers tab retourne *vrai* si tab contient que des entiers positifs ou nuls et *faux* sinon.
2. SOMME(tab_1,tab_2) qui étant donnés deux tableaux d'entiers tab_1 et tab_2 de même tailles retourne le tableau obtenu en sommant les cases de tab_1 et tab_2 deux à deux.

Correction.

1.
```
fonction EST_POSITIF(tab)
    pour i allant de 0 à longueur(tab) − 1 faire
        si tab[i] < 0 alors
            retourne Faux
        fin si
    fin pour
    retourne Vrai
fin fonction
```

2.
```
fonction SOMME(tab1, tab2)
    n ← longueur(tab1)
    créer un tableau res de taille n
    pour i allant de 0 à n − 1 faire
        res[i] ← tab1[i] + tab2[i]
    fin pour
    retourne res
fin fonction
```

2. Recherche d'un élément dans un tableau non trié

Définition 1. La complexité en temps *dans le pire des cas* d'un algorithme mesure le temps maximal que mettra un algorithme à finir pour une entrée de taille fixe.

Dans ce chapitre les calculs de complexité dans se feront en comptant le nombre de lectures/écritures dans le tableau ainsi que le nombre de comparaisons entre différent éléments du tableau. Pour un algorithme donné prenant en entré en tableau, nous noterons a_n et c_n le nombre de lectures/écritures et comparaison dans le pire des cas lorsque le tableau est de taille n. Le cout d'une lecture/écriture ou d'une comparaison étant en $O(1)$, nous obtiendrons ainsi que la complexité de l'algorithme dans le pire des cas est en $O(a_n + c_n)$.

Exercice.

1. Ecrire un algorithme RECHERCHE(tab, x) qui teste si un l'entier x est présent dans le tableau tab

Supposons aue tab soit un tableau de taille n.

2. Pour quelle(s) valeurs de x somme nous dans le pire des cas ? Que vallent les nombres a_n et c_n ?
3. En déduire la complexité de l'algorithme.

Correction.

1.

```
fonction RECHERCHE(tab, x)
    pour i allant de 0 à longueur(tab) − 1 faire
        si tab[i] = x alors
            retourne Vrai
        fin si
    fin pour
    retourne Faux
fin fonction
```

2. Le pire des cas correspond au cas où x n'est pas présent dans le tableau tab. Nous avons alors $a_n = c_n = n$.
3. La complexité dans le pire des cas est en $(a_n + b_n) \times O(1) = O(a_n + b_n) = O(2n) = O(n)$.

3. Recherche d'un élément dans un tableau trié

De nombreux problème algorithmiques peuvent menener à la recherhe d'un élément dans un ensemble trié.

Définition 2. La technique algorithmique *diviser pour régner* consiste à

1. Diviser : découper un problème initial en plusieurs sous-problèmes ;
2. Régner : résoudre les sous-problèmes ;
3. Combiner : déterminer une solution au problème inital à partir des réponses des sous-problèmes.

L'algorithme de recherche par dichotomie utilise la technique diviser pour régner similaire à celle que nous utilisons pour rechercher un mot au sein d'un dictionnaire. Afin de fixer le contexte supposons que l'on recherhce l'entier 5 dans le tableau trié

$$[1, 1, 2, 4, 6, 7, 10, 12, 14] = [1, 1, 2, 4] \cup [\mathbf{6}] \cup [7, 10, 12, 14]$$

L'entier se trouvant au milieu du tableau est 6 qui est plus grand que 5. Nous poursuivons ainsi notre recherche dans le sous-tableau gauche $[1, 1, 2, 4] = [1] \cup [\mathbf{1}] \cup [2, 4]$. Comme nous avons $1 < 5$ nous devons chercher 5 dans $[2, 4] = [\,] \cup [\mathbf{2}] \cup [4]$. De $2 < 5$ nous obtenons que 5 est à chercher dans $[4] = [\,] \cup [4] \cup [\,]$. Cqui est vide. Nous concluons donc que 5 n'est pas présent dans le tableau initial.

Théorème 3 (Master théorème)**.** Considérons un algorithme récursif utilisant la technique diviser pour régner qui découpe un problème initial de taille n en a sous problèmes de taille n/b. On note $f(n)$ le coût hors appels récursifs d'un problème de taille n (partage en sous problèmes, recombinaison des sous problèmes, ...). On suppose que $f(n)$ est en $O(n^d)$. Le cout de cet algorithme pour résoudre un probème de taille n est donc

$$C(n) = aC\left(\frac{n}{b}\right) + O(n^d),$$

et on a

$$\begin{aligned} &C(n) \in O(n^{\log_b(a)}) && \text{si } d < \log_b(a), \\ &C(n) \in O(n^d \log(n)) && \text{si } d = \log_b(a), \\ &C(n) \in O(n^d) && \text{si } d > \log_b(a). \end{aligned}$$

Exercice.

1. Ecrire un algorithme récursif pour la recherche par dichotomie d'un entier x dans le tableau tab de taille n.
2. Pour quelle(s) valeur(s) de x le pire des cas est-il atteint (pour a_n et c_n) ?

Correction.

1.

```
fonction DICHOTOMIE(tab, x)
    retourne DICHOTOMIE_REC(tab, 0, longueur(tab) − 1, x)
fin fonction
```

```
fonction DICHOTOMIE_REC(tab, g, d, x)
    si g > d alors
        retourne Faux
    fin si
    m ← ⌊(g+d)/2⌋
    si tab[m] = x alors
        retourne Vrai
    sinon si tab[m] < x alors
        retourne DICHOTOMIE_REC(tab, m + 1, d, x)
    sinon
        retourne DICHOTOMIE_REC(tab, g, m − 1, x)
    fin si
fin fonction
```

2. Le pire des cas est atteint lorsque x n'est pas présent dans le tableau tab.

Proposition 4. La compléxité dans le pire des cas de l'algorithme de recherche par dichotomie exécuté sur un tableau trié de taille n est en $O(\log n)$.

Démonstration. En reprenant les notations du master théorème nous avons $a = 1$, $b = 2$. Pour $f(n)$ nous avons 3 lectures/écritures. D'où $f(n) = O(1) = O(n^0)$ et donc $d = 0$. On a $a = b^0$ et donc $\log_b(a) = 0 = d$. Par le master théorème la complexité de l'algorithme de recherche par dichotomie est en $O(n^d \log(n)) = O(\log(n))$. □

4. Tri par séléction

Soit *tab* un tableau de taille n. Le principe du tri par sélection est :

1. étape 0 : chercher le plus petit élement de *tab* et l'échanger avec celui d'indice 0 ;
2. étape 1 : chercher le second plus petit élement de *tab* et l'échanger avec celui d'indice 1 ;
3. ...

Ainsi à l'étape i, les élements de tab d'indices $0, \dots, i-1$ sont triés et sont les i plus petit éléments de *tab*. Nous cherchons alors le plus petit élément de la partie non triée (indices enter i et $n-1$) et nous l'échangeons avec celui d'indice i.

Exercice.

1. Appliquer à la main le tri par séléction sur le tableau $[5, 2, 3, 1, 4, 1, 0, 3]$.

2. Écrire l'algorithme de tri par séléction.

Correction.

1. Voici les tableaux obtenus aprés les différentes étapes

$$\begin{aligned}
\text{etape } 0 &: \quad [0, 2, 3, 1, 4, 1, 5, 3] \\
\text{etape } 1 &: \quad [0, 1, 3, 2, 4, 1, 5, 3] \\
\text{etape } 2 &: \quad [0, 1, 1, 2, 4, 3, 5, 3] \\
\text{etape } 3 &: \quad [0, 1, 1, 2, 4, 3, 5, 3] \\
\text{etape } 4 &: \quad [0, 1, 1, 2, 3, 4, 5, 3] \\
\text{etape } 5 &: \quad [0, 1, 1, 2, 3, 3, 5, 4] \\
\text{etape } 6 &: \quad [0, 1, 1, 2, 3, 3, 4, 5]
\end{aligned}$$

2.

```
fonction TRI_SELECTION(tab)
    pour i de 0 à longueur(tab) − 2 faire
        i_min ← i
        x_min ← tab[i_min]
        pour j de i + 1 à longueur(tab) − 1 faire
            x ← tab[j]
            si x < x_min alors
                i_min ← j
                x_min ← x
            fin si
        fin pour
        si i_min ≠ i alors
            tab[i_min] ← tab[i]
            tab[i] ← x_min
        fin si
    fin pour
fin fonction
```

Proposition 5. La compléxité dans le pire des de l'algorithme de tri par sélection exécuté sur un tableau de taille n est en $O(n^2)$.

Démonstration. Pour $n \leq 1$, on a $a_n = c_n = 0$. Supposons $n \geq 2$. Quel que soit le tableau de taille n, la boucle i de l'algorithme TRI_SELECTION nécessite $n-i-1$ comparaisons. Concernant

le nombre de lectures/écritures le pire des cas est atteint lorsque à l'issu de la boucle j on a toujours $i_{min} \neq i$. C'est le cas en particulier lorsque tab est le tableau $[n, 1, 2, \ldots, n-1]$. La boucle i nécessite alors $n + 3 - i$ lectures/écritures. Mous obtenons ainsi

$$c_n = \sum_{i=0}^{n-1} n - i - 1 = \sum_{k=0}^{n-1} k = \frac{(n-1)n}{2}$$
$$a_n = \sum_{i=0}^{n-1} n - i + 3 = 3n + \sum_{i=0}^{n-1} n - i = 3n + \sum_{k=1}^{n} k = 3n + \frac{n(n+1)}{2}.$$

On a dons que a_n et b_n sont en $O(n^2)$. La compléxité dans le pire des cas de l'algorithme de tri par séléction est donc en $O(n^2)$. □

5. Tri fusion

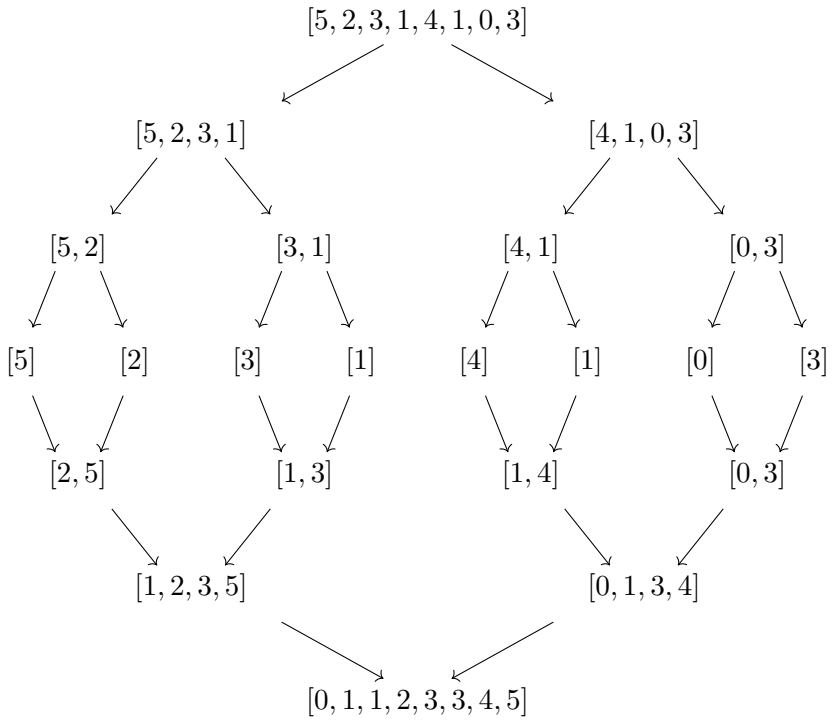
Le tri fusion est un algorithme récursif exploitant la technique diviser pour regner. Le tableau initial est coupé en deux sous tableaux de tailles comparables. Chacun de ces sous tableaux est alors triés à l'aide du tri fusion. Finalement une fusion de ces deux sous tableaux triés est effectuée.

Exercice.

1. Appliquer à la main le tri fusion sur le tableau $[5, 2, 3, 1, 4, 1, 0, 3]$.
2. Ecrire un algortihme FUSION(tab,g,m,d) qui fusionne les deux sous tableaux $tab[g, \ldots, m]$ et $tab[m+1, \ldots, d]$ de tab. Le tableau fusionné sera alors le sous tableau $tab[g, \ldots, d]$.
3. Ecrire l'algorithme de tri fusion.

Correction.

1.



2.

```
fonction FUSION(tab,g,m,d)
    Créer un tableau temp de longueur d − g + 1o.
    i_g ← g
    i_d ← m + 1
    j ← 0
    tant que i_g ≤ m et i_d ≤ d faire
        x_g ← tab[i_g]
        x_d ← tab[i_d]
        si x_g <= x_d alors
            temp[j] ← x_g
            i_g ← i_g + 1
        sinon
            temp[j] ← x_d
            i_d ← i_d + 1
        fin si
        j ← j + 1
    fin tant que
    si i_g ≤ m alors
        tant que i_g ≤ m faire
            temp[j] ← tab[i_g]
            i_g ← i_g + 1
            j ← j + 1
        fin tant que
    sinon
        tant que i_d ≤ d faire
            temp[j] ← tab[i_d]
            i_d ← i_d + 1
            j ← j + 1
        fin tant que
    fin si
    pour j de 0 à d − g faire
        tab[j + g] = temp[j]
    fin pour
fin fonction
```

3.

```
fonction TRI_FUSION(tab)
    TRI_FUSION_REC(tab, 0, longueur(tab) − 1)
fin fonction
```

```
fonction TRI_FUSION_REC(tab,g,d)
    si g < d alors
        m ← ⌊(g+d)/2⌋
        TRI_FUSION_REC(tab,g,m)
        TRI_FUSION_REC(tab,m + 1,d)
        FUSION(tab,g,m,d)
    fin si
fin fonction
```

Proposition 6. La compléxité dans le pire des cas de l'algorithme du tri fusion exécuté sur un tableau de taille n est en $O(n \log(n))$.

Démonstration. Avec les notations du master théorème nous avons $a = 2$, $b = 2$. Pour $f(n)$ l'algorithme de fusion est à considérer. Comme il y a au pire $6n$ lectures/écritures et n comparaisons on a $f(n) \in O(n)$ et donc $d = 1$. De $d = \log_b(a)$ le master théormème implique que la compléxité du tri fusion est en $O(n^d \log(n)) = O(n \log(n))$. □

6. Tri rapide

Le tri rapide est un autre tri basé sur la technique *divisé pour régner*. Le principe de ce tri est le suivant :

1. Choisir un élément du tableau qu'on appelera pivot ; la séléction du pivot peut être deterministe (premier élément, dernier élément, ...) ou bien aléatoire.
2. Effectuer le pivotage du tableau. Cela consiste à séparer les éléments inferieurs (ou égale) et supérieurs (strictement) au pivot. Les éléments inférieurs au pivot se retrouve alors au début du tableau tandis que les éléments supérieurs se retrouvent en fin de tableau. Quand à lui le pivot se retrouve entre les deux groupes.
3. Recommencer avec les deux sous tableaux consistitués des éléments inférieurs puis supérieurs au pivot.

Pour le pivotage une méthode consiste à échanger le pivot avec le dernier élément du tableau. On lit alors le tableau de la gauche vers la droite. Notons i_k le nombre d'éléments inférieurs au pivot placés au début du tableau. On a donc $i_0 = 0$. A l'étape $k + 1$ on lit $k + 1$ ème x élement du tableau. Si x est inférieur ou égale au pivot on l'échange avec celui à la position i_k et on a alors $i_{k+1} = i_k + 1$. Si x est supérieur au pivot on ne fait rien et on a $i_{k+1} = i_k$.

– si l'élément lu est inférieur au pivot on le place aprés les autres éléments inférierus en début de tableau

– sinon on le place devant échanger le pivot avec le dernier élément du tableau. Puis on place au début du tableau les éléments inférieurs au pivot, puis le pivot.

Exercice.

1. Appliquer à la main l'algorithme de tri rapide sur le tableau $[5, 2, 3, 1, 4, 1, 0, 3]$ en choisissant systématiquement l'élément de plus haut indice comme pivot.

2. Écrire une fonction PIVOTAGE(*tab*,*g*,*d*,*p*) qui effectue un pivotage du sous tableau

$$[tab[g], \ldots, tab[d]]$$

de *tab* avec un pivot se trouvant à la position $p \in [g, d]$ et retourne la position du pivot dans le nouveau tableau.

3. Écrire l'algorithme du tri rapide en choisissant systématiquement l'élément de plus haut indice comme pivot.

4. Comparer le nombre de comparaisons et le nombre de lectures/écritures lors de l'éxécution de l'algorithme PIVOTAGE.

5. Justifier que le calcul de compléxite du tri rapide ne se fasse en considérant seulement le nombre de comparaisons.

Correction.

1. Le pivotage de $[5,2,3,1,4,1,0,3]$ donne $[2,3,1,1,0,\mathbf{3},4,5]$ et donc $[2,3,1,1,0,3,4,5]$
Le pivotage de $[2,3,1,1,0]$ donne $[\mathbf{0},3,1,1,2]$ et donc $[0,3,1,1,2,3,4,5]$
Le pivotage de $[3,1,1,2]$ donne $[1,1,\mathbf{2},3]$ et donc $[0,1,1,2,3,3,4,5]$
Le pivotage de $[1,1]$ donne $[1,\mathbf{1}]$ et donc $[0,1,1,2,3,3,4,5]$
Le pivotage de $[1]$ donne $[\mathbf{1}]$ et donc $[0,1,1,2,3,3,4,5]$
Le pivotage de $[3]$ donne $[\mathbf{3}]$ et donc $[0,1,1,2,3,3,4,5]$
Le pivotage de $[2]$ donne $[\mathbf{2}]$ et donc $[0,1,1,2,3,3,4,5]$
Le pivotage de $[4,5]$ donne $[4,\mathbf{5}]$ et donc $[0,1,1,2,3,3,4,5]$
Le pivotage de $[4]$ donne $[\mathbf{4}]$ et donc $[0,1,1,2,3,3,4,5]$

2.

```
fonction PIVOTAGE(tab, g, d, p)
    pivot ← tab[p]
    tab[p] ← tab[d]
    tab[d] ← pivot
    i ← g
    pour j de g à d faire
        t ← tab[j]
        si t ≤ pivot alors
            tab[j] ← tab[i]
            tab[i] ← t
            i ← i + 1
        fin si
    fin pour
    retourne i
fin fonction
```

3.

```
fonction TRI_RAPIDE(tab)
    TRI_RAPIDE_REC(tab, 0, longueur(tab) − 1)
fin fonction
```

```
fonction TRI_RAPIDE_REC(tab, g, d)
    si g < d alors
        p ← PIVOTAGE(tab,g,d,d)
        TRI_RAPIDE_REC(tab, g, p − 1)
        TRI_RAPIDE_REC(tab, p + 1, d)
    fin si
fin fonction
```

4. Lors de l'appel de PIVOTAGE(tab, g, d, p), il y a exactement $d-g+1$ comparaisons. Il y a au minimum 4 et au maximum $4+4(d-g+1)$ lectures/écritures.

5. Pour un tableau tab notons a_{tab} et c_{tab} le nombre de lectures/écritures et le nombre de comparaisons lors de éxécution de l'algorithme TRI_RAPIDE. Par ce aui précède, nous avons $a_{tab} \leq 4 + 4\,c_{tab}$ et donc $a_{tab} + c_{tab} \leq 4 + 5\,c_{tab}$, d'où $a_{tab} + c_{tab} \in O(c_{tab})$.

Proposition 7. La compléxité dans le pire des cas de l'algorithme du tri rapide pour trié un tableau de taille n est en $O(n^2)$.

Démonstration. Le pire des cas est obtenu lorsque le pivot est toujours le plus petit ou le plus grand élément du tableau. Avec notre choix de pivot ceci arrive, par exemple, lorsque le tableau est déjà trié. Ainsi l'appel de l'algorithme de tri rapide sur un tableau de taille n nécessite n comparaison provenant du pivotage et c_{n-1} provenant du tri du sous tableau de taille $n-1$

constitué des entrées inférieurs au pivot. Et donc, pour $n \geq 2$, nous obtenons $c_n = n + c_{n-1}$. De $c_0 = c_1 = 0$, nous obtenons

$$c_n = \sum_{i=2}^{n} i = \frac{n(n+1)}{2} - 1 \in O(n^2). \qquad \square$$

Définition 8. La compléxité en temps en moyenne d'un algorithme mesure le temps moyen que met l'algortihme lorsque il est lancé sur une entrée tirée aléatoirement parmi toutes les entrées possibles.

Proposition 9. La compléxité en moyenne de l'algorithme de tri rapide pour trié un tableau de taille n est en $O(n \log n)$.

Démonstration. Notons C_n le nombre moyen de comparaisnn entre éléments du tableau lors de l'éxecution du tri rapide sur un tableau de longueur n. Comme les tableaux de longueurs 0 et 1 sont déjà triés nous avons $C_0 = C_1 = 0$. Traitons le cas d'un tableau de taille $n \geq 2$. Le pivotage d'un tableau de taille n nécessite n comparaisons nous avons Soit *tab* un tableau de taille n tiré aléatoirement selon une loi uniforme. Aprés pivotage, le pivot se retouve à la position p uniformément répartie entre 0 et $n-1$. Pour trié *tab*, en plus du pivotage, nous appliquons l'algorithme au sous tableau d'indices $[0, p-1]$ et celui d'indices $[p+1, n-1]$ pour un coût respectif de C_p et C_{n-p-1}. Le coût moyen pour trié un tableau de taille n est donc

$$\begin{aligned} C_n &= n + \frac{1}{n}\left(\sum_{p=0}^{n-1} C_p + C_{n-p-1}\right) = n + \frac{1}{n}\left(\sum_{p=0}^{n-1} C_p + \sum_{p=0}^{n-1} C_{n-p-1}\right) \\ &= n + \frac{2}{n}\sum_{p=0}^{n-1} C_p. \end{aligned}$$

En multipliant par n on obtient

$$\begin{aligned} n\,C_n &= n^2 + 2\sum_{p=0}^{n-1} C_p = n^2 + 2\,C_{n-1} + 2\sum_{p=0}^{n-2} C_p \\ &= n^2 + 2\,C_{n-1} + (n-1)\,C_{n-1} - (n-1)^2 \\ &= 2n - 1 + (n+1)C_{n-1}. \end{aligned}$$

Ainsi nous avons obtenu

$$\frac{C_n}{n+1} = \frac{C_{n-1}}{n} + \frac{2n-1}{n(n+1)} = \sum_{k=2}^{n} \frac{2k-1}{k(k+1)}.$$

La somme peut se réécrire

$$\begin{aligned} \sum_{k=2}^{n} \frac{2k-1}{k(k+1)} &= \sum_{k=2}^{n} \frac{2k}{k(k+1)} - \frac{1}{k(k+1)} = \sum_{k=2}^{n} \frac{1}{k+1} - \sum_{k=2}^{n} \frac{1}{k^2+k} \\ &= \sum_{k=3}^{n+1} \frac{1}{k} - \sum_{k=2}^{n} \frac{1}{k^2+k}. \end{aligned}$$

La série $T_n = \displaystyle\sum_{k=1}^{n} \frac{1}{k^2+k}$ est à terme positif et dominée apr la série de Riemann $\sum_{k=1}^{n} \frac{1}{k^2}$ qui converge lorsque n tend vers $+\infty$. Nous avons donc $T_n \in O(1)$. Etudions maintenant la série

$S_n = \sum_{k=3}^{n+1} \frac{1}{k}$. Comme $t \mapsto \frac{1}{t}$ est décroissante, pour $k \geq 2$, nous avons

$$\int_{k-1}^{k} \frac{1}{t}\ \mathrm{d}t \geq \int_{k-1}^{k} \frac{1}{k}\ \mathrm{d}t = \frac{1}{k},$$

et donc

$$S_n = \sum_{k=3}^{n+1} \frac{1}{k}\ \leq\ \sum_{k=3}^{n+1} \int_{k-1}^{k} \frac{1}{t}\ \mathrm{d}t\ =\ \int_{1}^{n+1} \frac{1}{t} \mathrm{d}t = \ln(n+1).$$

Comme le quotient $\ln(n+1)/\ln(n)$ converge vers 1 avec n, nous avons $S_n \in O(\ln(n))$. Nous obtenons ainsi $\frac{1}{n+1} C_n \in O(\ln n)$ et donc $C_n \in O(n \ln n)$. □