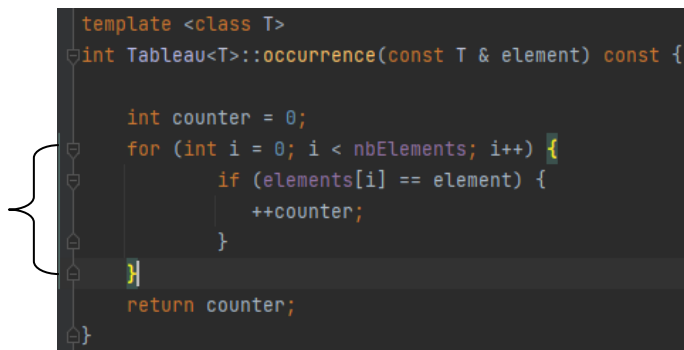


# ANALYSE DE LA COMPLEXITÉ TEMPORELLE (PIRE CAS) EN NOTATION GRAND O

Jean Marie HABWINTAHE - HABJ15017805

Jabes ADASZ - ADAJ14048000

## COMPLEXITE DE LA FONCTION OCCURRENCE



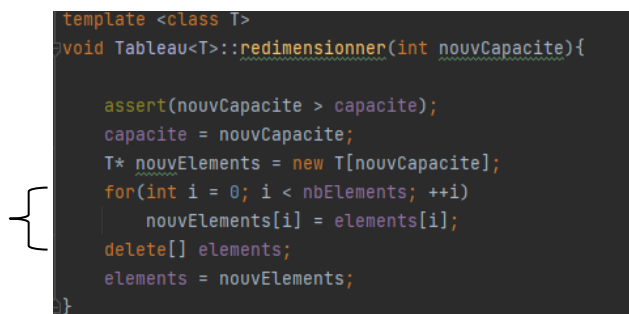
```
template <class T>
int Tableau<T>::occurrence(const T & element) const {
    int counter = 0;
    for (int i = 0; i < nbElements; i++) {
        if (elements[i] == element) {
            ++counter;
        }
    }
    return counter;
}
```

A bracket on the left side of the code block groups the lines from the `for` loop to the closing brace of the function, indicating that this section of the code is responsible for the  $O(n)$  complexity.

La complexité de cette fonction est  $O(n)$ . En effet, la boucle `for` délimité par l'accolade extérieure a la complexité  $O(n)$  car il parcourt tout le tableau. les autres lignes sont de complexité constante.

**$O(\text{Constante} + n + \text{constante}) = O(n)$ .**

## COMPLEXITE DE LA FONCTION INSERER



```
template <class T>
void Tableau<T>::redimensionner(int nouvCapacite){
    assert(nouvCapacite > capacite);
    capacite = nouvCapacite;
    T* nouvElements = new T[nouvCapacite];
    for(int i = 0; i < nbElements; ++i)
        nouvElements[i] = elements[i];
    delete[] elements;
    elements = nouvElements;
}
```

A bracket on the left side of the code block groups the lines from the `for` loop to the `delete[]` statement, indicating that this section of the code is responsible for the  $O(n)$  complexity.

```

template <class T>
void Tableau<T>::insérer(const T & element, int index) {
    if(nbElements == capacite)
        redimensionner( nouvCapacite: capacite * 2);
    for(int i = nbElements; i > index; --i){
        elements[i] = elements[i-1];
    }
    elements[index] = element;
    ++nbElements;
}

```

La fonction insérer appelle la fonction redimensionner. Or, l'accolade qui délimite la boucle for de la fonction redimensionner à la complexité  $O(n)$ . Comme cette fonction est appelée dans la fonction insérer, cette complexité se répercute aussi dans la fonction insérer (cfr la ligne délimitée par la petite accolade dans la fonction insérer). La deuxième accolade, la grande, dans la fonction insérer a aussi la complexité égale à  $n$ . Les autres lignes de la fonction ont une complexité constante.

**$O(\text{constante}+n+n+\text{constante}) = O(n)$ .**

## COMPLEXITE DE LA FONCTION UNIQUE

```

template<class T>
void Tableau<T>::unique() {
    for (int i = 0; i < nbElements; i++) {
        for (int j = i + 1; j < nbElements; j++) {
            if (elements[i] == elements[j]) {
                for (int k = j; k < nbElements - 1; k++) {
                    elements[k] = elements[k + 1];
                }
                nbElements--;
                j--;
            }
        }
    }
}

```

Si l'on considère mon implémentation, la pire situation peut se produire si aucun doublon n'existe dans le tableau.

Dans ce cas, la boucle externe, délimitée par la grande accolade, va avoir la complexité  $O(n)$ . De même, la boucle délimitée par l'accolade du milieu (qui a comme indice  $j$ ) va avoir la complexité  $O(n-1)$ .


La 3 -ème boucle, délimitée par la plus petite accolade (qui comme indice  $k$ ), ne va jamais s'exécuter.

**Dans ce cas, la complexité sera de  $O(n(n-1)) = O(n^2-n) = O(n^2)$ .**

## COMPLEXITE DE LA FONCTION SUGGESTION



```
Tableau<std::string> Membre::retournerListeFavoris(const Tableau<std::string> &liste) const {  
    Tableau<std::string> lesFavoris = liste;  
    for (int i = 0; i < lesFavoris.taille(); i++) {  
        for (int j = 0; j < favs.taille(); j++) {  
            if (lesFavoris[i] == favs[j]) {  
                lesFavoris.enlever(index: i);  
            }  
        }  
    }  
    return lesFavoris;  
}
```



```
Tableau<std::string> Membre::suggestion(const Membre &autre) const {  
    Tableau<std::string> lesFavorisAutre = autre.favs;  
    Tableau<std::string> listeFavoris = retournerListeFavoris(liste: lesFavorisAutre);  
    return listeFavoris;  
}
```

La fonction suggestion appelle la fonction retournerListeFavoris. Cette dernière a deux boucles imbriquées délimitées par les accolades comme le montre la capture d'écran de la fonction retournerListeFavoris. La taille des favoris du membre courant est  $m$  tandis que la taille des favoris de l'autre membre est  $d$ . Donc la complexité liée à cette fonction est  $O(m \times d)$ .

La fonction suggestion appelle la fonction retournerListeFavoris à la ligne pointée par la flèche ci-haut et il vient avec sa complexité à savoir  $O(m \times d)$ . Dans la fonction suggestion, les autres lignes comportent une complexité constante.

**Donc la complexité de la fonction suggestion est  $O(\text{constante} + m \times d + \text{constante}) = O(m \times d)$ .**