Complexité d’un algorithme

Soit f(n) une fonction de n.

On dira qu'un algorithme est en **O(f(n))** si son temps d'exécution est proportionnel à f(n).

On dira qu'un algorithme est en O(f(n)) s'il existe deux nombres positifs constants k et n_0 tels que $T(n) \leq k*f(n)$ quand $n = n_0$.

La fonction f(n) est appelée l'**ordre de complexité** de l'algorithme

Les cas les plus fréquemment rencontrés sont les suivants :

O(1) : durée indépendante de n
O(log n) : complexité logarithmique
O(n) : complexité linéaire
O(n * log n) : complexité quasi-linéaire
O(n^2) : complexité quadratique
O(n^3) : complexité cubique
O(n^d) : complexité polynomiale
O(2^n) : complexité exponentielle
O(n!) : complexité factorielle

n ↓	1	log n	n	n*log n	n^2	n^3	2^n	n!
10	1	3	10	30	100	1000	1000	3628800
20	1	4	20	80	400	8000	1000000	$2*10^{18}$
100	1	7	100	700	10000	1000000	10^{30}	...
1000	1	10	1000	10000	10^6	10^9	10^{300}	
10000	1	13	10000	130000	10^8	10^{12}	...	
10^6	1	20	1000000	20000000	10^{12}	10^{18}		
10^9	1	30	10^9	$30*10^9$	10^{18}	10^{27}		

Le facteur de proportionnalité est négligé.
Il ne sera utilisé que pour départager deux algorithmes qui ont le même ordre de complexité.

O(n/2)

O(n)

O(2n)

O(n)

```
public boolean contient(int unEntier){

    for (int i = 0; i < this.nombreDEntiers; i++) {

        if(this.tableDEntiers[i]==unEntier)
            return true;
    }

    return false;
}
```

Dans le meilleur des cas :

O(1)

3	9	8	5	1

```
public boolean contient(int unEntier){

    for (int i = 0; i < this.nombreDEntiers; i++) {

        if(this.tableDEntiers[i]==unEntier)
            return true;
    }

    return false;
}
```

Dans le pire des cas :

O(n)

```
public boolean contient(int unEntier){

   for (int i = 0; i < this.nombreDEntiers; i++) {

       if(this.tableDEntiers[i]==unEntier)
          return true;
   }

   return false;
}
```

Coût moyen:

O(n/2)

3	9	8	5	1

```
public boolean contient(int unEntier){

   for (int i = 0; i < this.nombreDEntiers; i++) {

       if(this.tableDEntiers[i]==unEntier)
          return true;
   }

   return false;
}
```

En ignorant le facteur de proportionnalité:

O(n)

3	9	8	5	1

```
public boolean contient(int unEntier){
    return trouverIndice(unEntier) != -1;
}
```

O(n)

O(n)

```
public boolean contientExAequo(){

   for (int i = 0; i < nombreDEntiers - 1; i++) {

       for (int j = i+1; j < nombreDEntiers; j++) {

           if(tableDEntiers[i]==tableDEntiers[j])

              return true;
       }
   }
   return false;
}
```

$O(n^2)$

```
public boolean supprimer(int unEntier){

    for (int i = 0; i < nombreDEntiers; i++) {

        if(tableDEntiers[i]==unEntier){

            for (int j = i; j < nombreDEntiers-1; j++)

                tableDEntiers[j]=tableDEntiers[j+1];

            nombreDEntiers--;
            return true;
        }
    }
    return false;
}
```

O(n)

```
public boolean supprimer(int unEntier){

    int indice = trouverIndice(unEntier);

    if(indice == -1)
        return false;

    supprimerALIndice(indice);

    return true;

}
```

O(n)

O(n)

O(n)

```
public void ajouter(int unEntier){

    agrandirTableSiPleine();
    tableDEntiers[nombreDEntiers]=unEntier;
    nombreDEntiers++;
}
```

O(n)

Coût amorti = 1!!!

Moyenne de n ajout

L'agrandissement de la table se fait tous les n ajouts si taille x 2