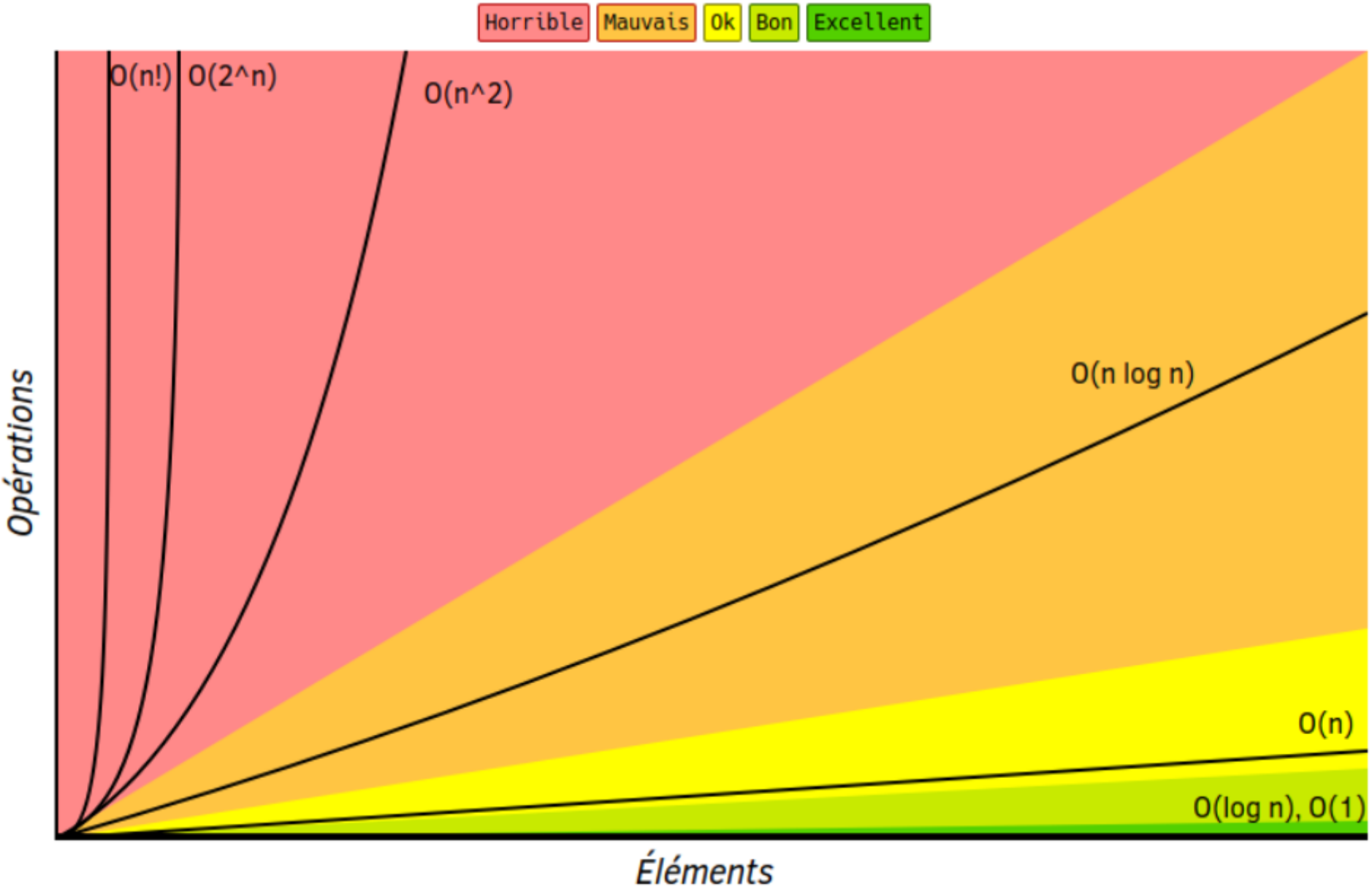
Complexité d’un algorithme



buzut.net/cours/computer-science/time-complexity

Ordre de complexité

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}		

Nombre de données

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}		

J'ai 1.000.000 d'éléments :

Si l'algorithme en O(1) prend 1 unité de temps, ☺

l'algorithme en O(logN) prendra 20 unités de temps, ☺

l'algorithme en O(N) prendra 1.000.000 unités de temps, ☺

l'algorithme en O(NlogN) prendra 20.000.000 unités de temps, ☺

l'algorithme en $O(N^2)$ prendra 10^{12} unités de temps. ☹

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)

Exemples

```
public class TableauNonTrieDEntiersV4 {

    private int[] t;
    private int nombreEntiers; // taille logique

    // Les nombrEntiers entiers occupent les
    // nombreEntiers premières cases du tableau!
    // PAS DE TROU!
    // L'ordre des entiers doit être conservé lors des
    // suppressions
    // Il faut agrandir la table si nécessaire
}
```

```
public boolean contient(int entier){

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

      if(t[i]==entier)
          return true;
   }

   return false;
}
```

Coût ?

```
public boolean contient(int entier){

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

        if(t[i]==entier)
            return true;
    }

    return false;
}
```

Dans le meilleur des cas :

O(1)

```
public boolean contient(int entier){

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

        if(t[i]==entier)
            return true;
    }

    return false;
}
```

Dans le pire des cas :

O(n)

```
public boolean contient(int entier){

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

        if(t[i]==entier)
            return true;
    }

    return false;
}
```

Coût moyen :

O(n/2)

3	9	8	5	1

```
public boolean contient(int entier){

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

        if(t[i]==entier)
            return true;
    }

    return false;
}
```

En ignorant le facteur de proportionnalité:

O(n)

3	9	8	5	1

```
private int trouverIndice(int entier){

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

        if(t[i]==unEntier)
            return i;
    }

    return -1;
}
```

O(n)

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

Coût?

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

O(n)

O(n)

```
public boolean contientExAequo(){

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

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

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

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

Coût?

```
public boolean contientExAequo(){

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

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

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

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

Coût?

$$1 + 2 + 3 + \ldots + n = \frac{n(n+1)}{2}$$

1x 2x 3x nx

```
public boolean contientExAequo(){

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

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

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

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

$O(n^2)$

```
public boolean supprimer(int entier){

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

        if(t[i]==entier){

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

                t[j]=t[j+1];

            nombreEntiers--;
            return true;
        }
    }
    return false;
}
```

O(n)

```
public boolean supprimer(int entier){

    int indice = trouverIndice(entier);

    if(indice == -1)
        return false;

    supprimerALIndice(indice);

    return true;

}
```

O(n)

O(n)

O(n)

```
public void ajouter(int entier){

    agrandirTableSiPleine();
    tableDEntiers[nombreEntiers]=entier;
    nombreEntiers++;
}
```

O(n)

Coût si agrandissement de table : O(N)

Coût si pas agrandissement de table : O(1)

```
public void ajouter(int entier){

    agrandirTableSiPleine();
    tableDEntiers[nombreEntiers]=entier;
    nombreEntiers++;
}
```

O(n)

Coût amorti = 1!!!

Moyenne de n ajout

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