**RÉPUBLIQUE ALGÉRIENNE DÉMOCRATIQUE ET POPULAIRE**

**MINISTERE DE L'ENSEIGNEMENT SUPERIEUR ET DE LA RECHERCHE SCIENTIQUE**

**UNIVERSITÉ DES SCIENCES ET DE LA TECHOLOGIE HOUARI BOUMEDIENE**


# Rapport TP 2 : Recherche d'un élément

**Réalisé par** : ATTARI LINA

BENAICHA CHAHINEZ

CHAIB AYOUB

MIRI MOHAMED WEAAM

# A. Recherche d'un élément

On étudie dans cet exercice la complexité d'un algorithme de recherche d'un élément x dans un ensemble A.

***1. Soit un tableau de n (n>=2) valeurs entières non triées***

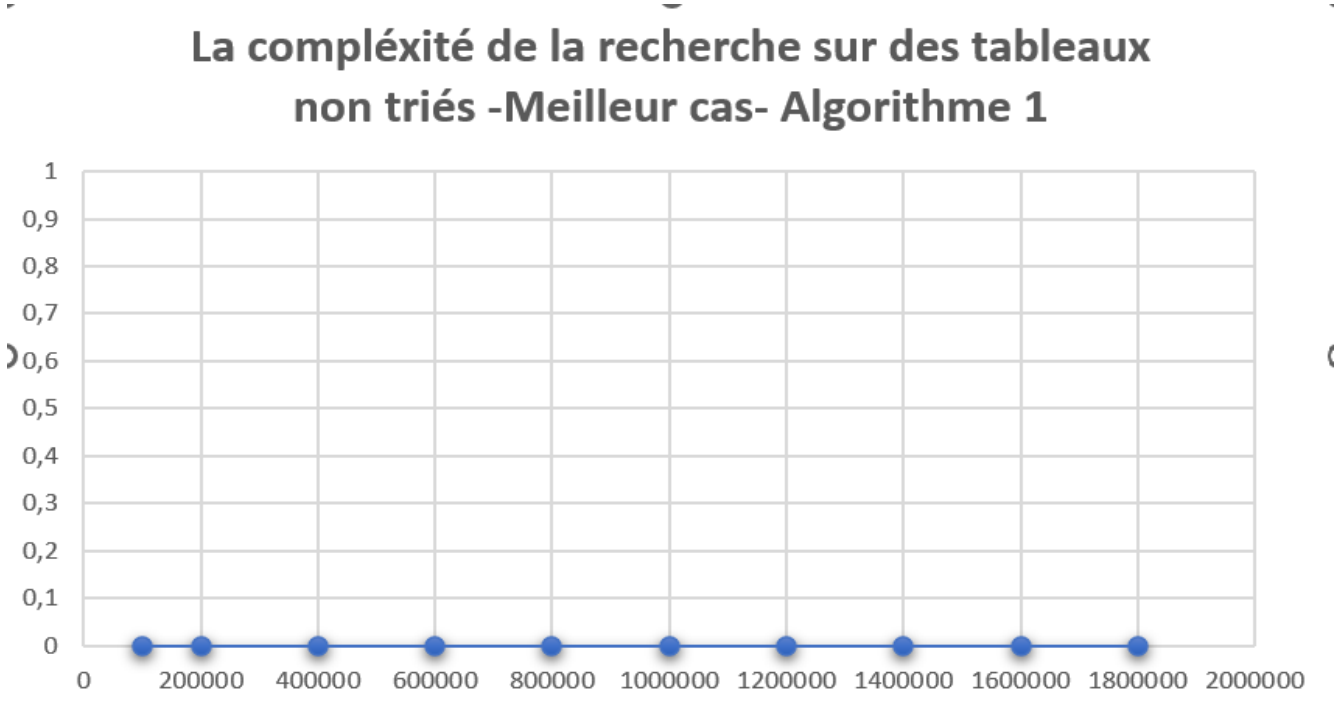
a. Écrire une fonction **rechElets_TabNonTriés** permettant de vérifier

L'existence d'une valeur x donnée.

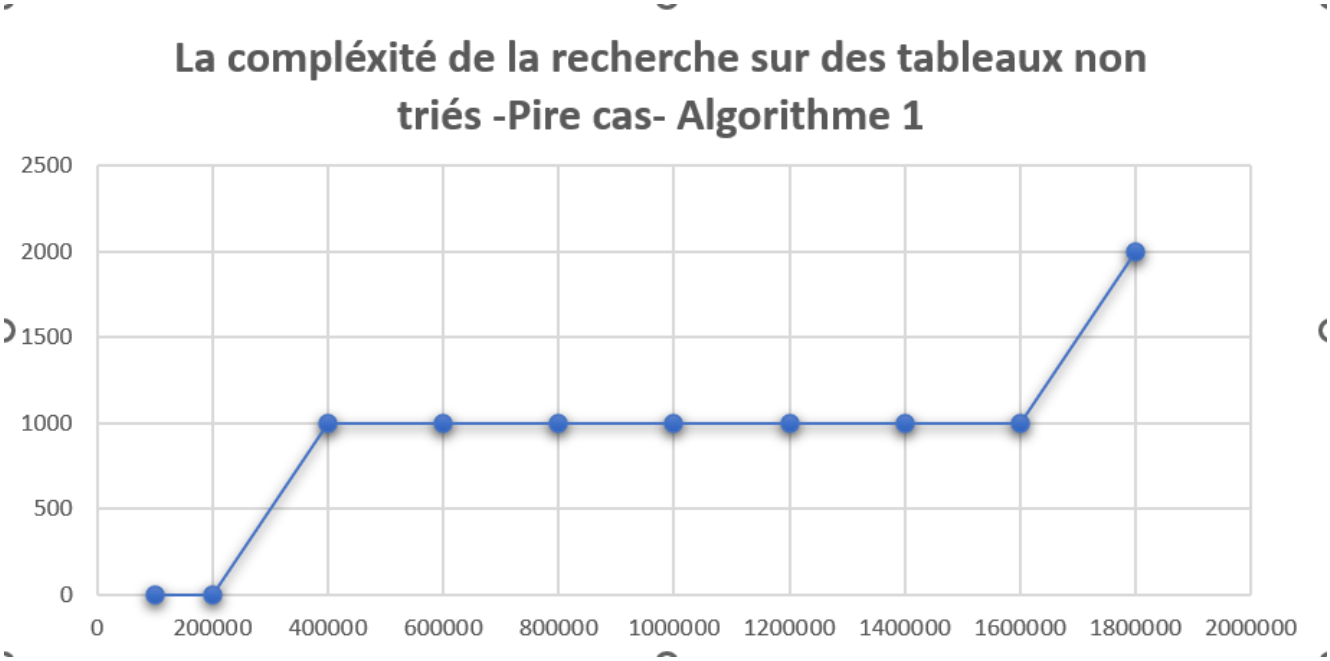
```
int rechElets_TabNonTries(int T[], int n, int x)
{
    for (int i = 0; i < n; i++)
    {
        if (T[i] == x)
            return 1;   // trouvé
    }
    return 0;   // non trouvé
}
```

**Complexité**

- **Meilleur cas :** O (1)
  (si x est à la première case)



- **Pire cas :** O(n)
  (si x est à la dernière case ou absent)




***2. Soit un tableau de n (n>=2) valeurs entières triées***

## 2.1. Recherche séquentielle

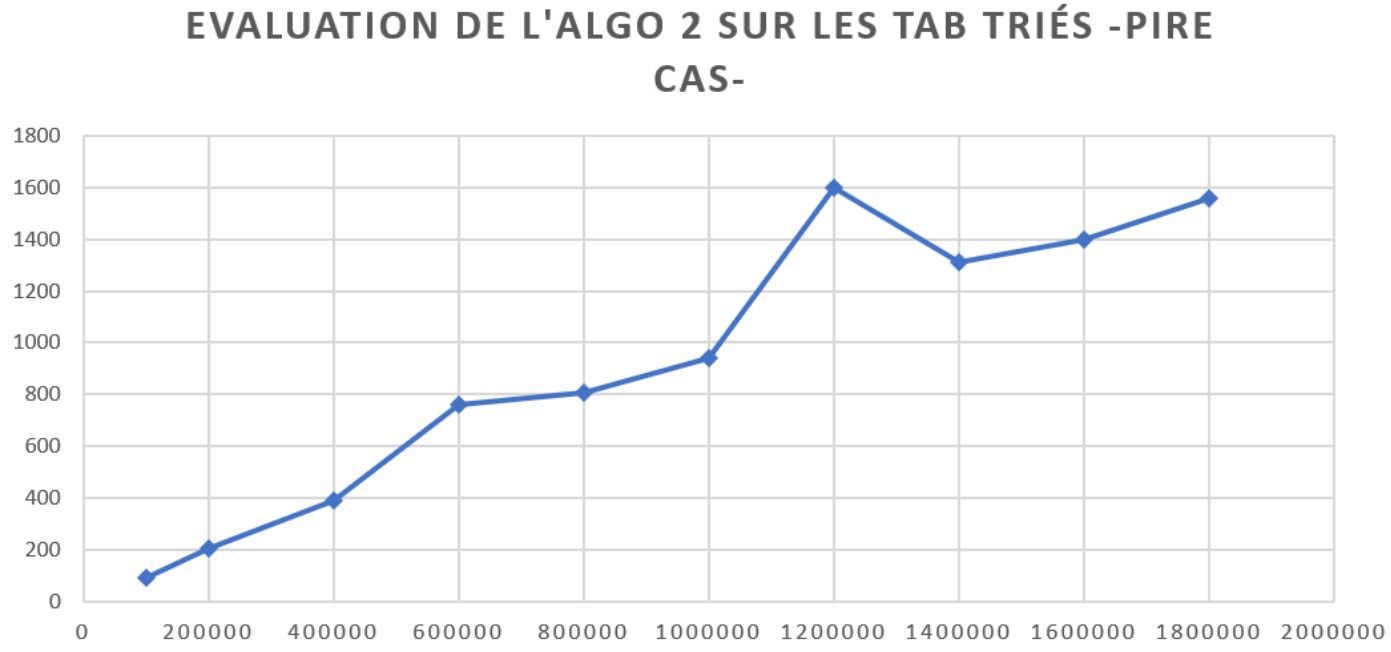
a. Écrire une fonction **rechElets_TabTriés** permettant de vérifier l'existence d'une valeur x donnée.

```
int rechElets_TabTries(int T[], int n, int x)
{
    for (int i = 0; i < n; i++)
    {
        if (T[i] == x)
            return 1;
        if (T[i] > x)   // optimisation car tableau trié
            return 0;
    }
    return 0;
}
```

## Complexité

- **Meilleur cas :** O (1)


- **Pire cas :** O(n)



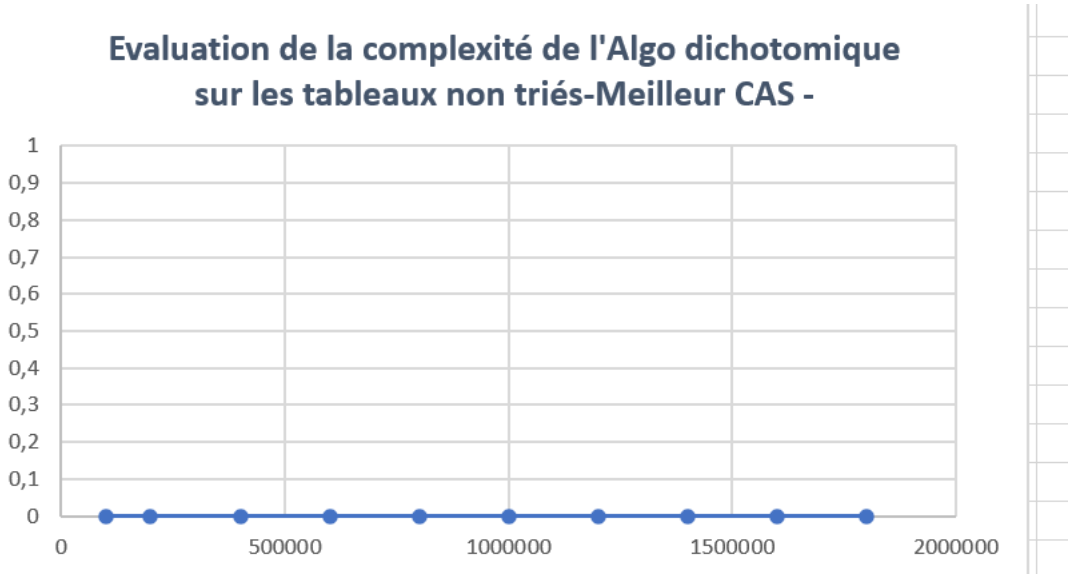
## 2.2. Recherche Dichotomique :

a. Écrire une fonction **rechElets_Dicho permettant** de vérifier l'existence d'une valeur x donnée en utilisant la méthode dichotomique.

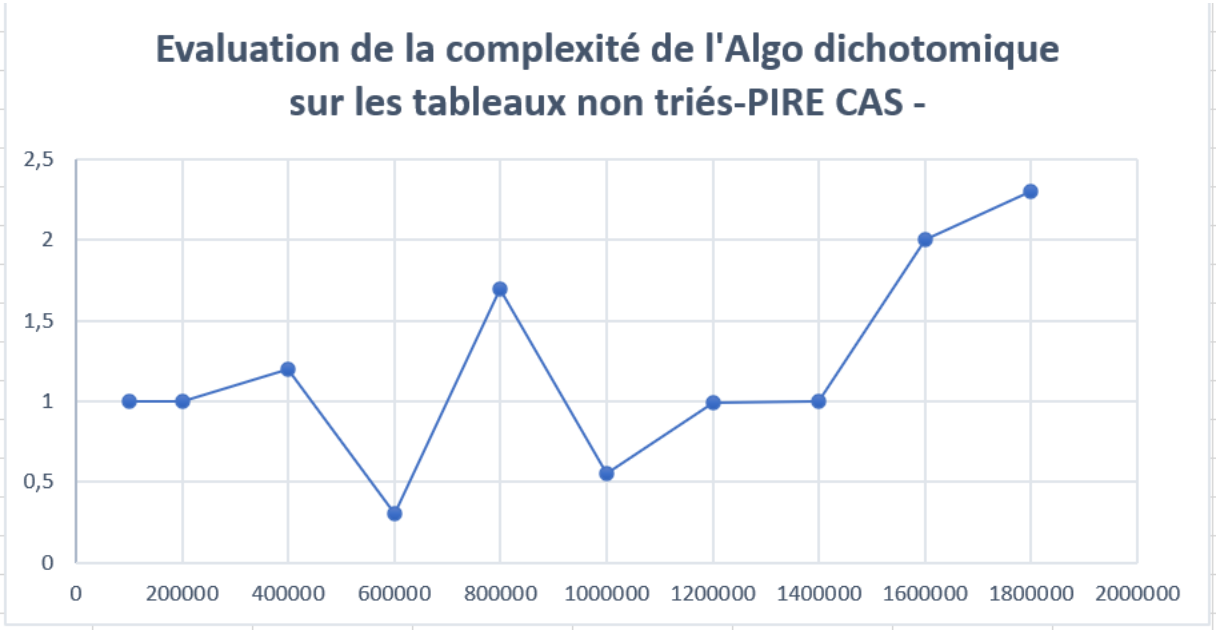
```
int rechElets_Dicho(int T[], int n, int x)
{
    int gauche = 0, droite = n - 1;

    while (gauche <= droite)
    {
        int milieu = (gauche + droite) / 2;

        if (T[milieu] == x)
            return 1;

        if (T[milieu] < x)
            gauche = milieu + 1;
        else
            droite = milieu - 1;
    }
    return 0;
}
```

**Complexité**

- **Meilleur cas:** O (1)



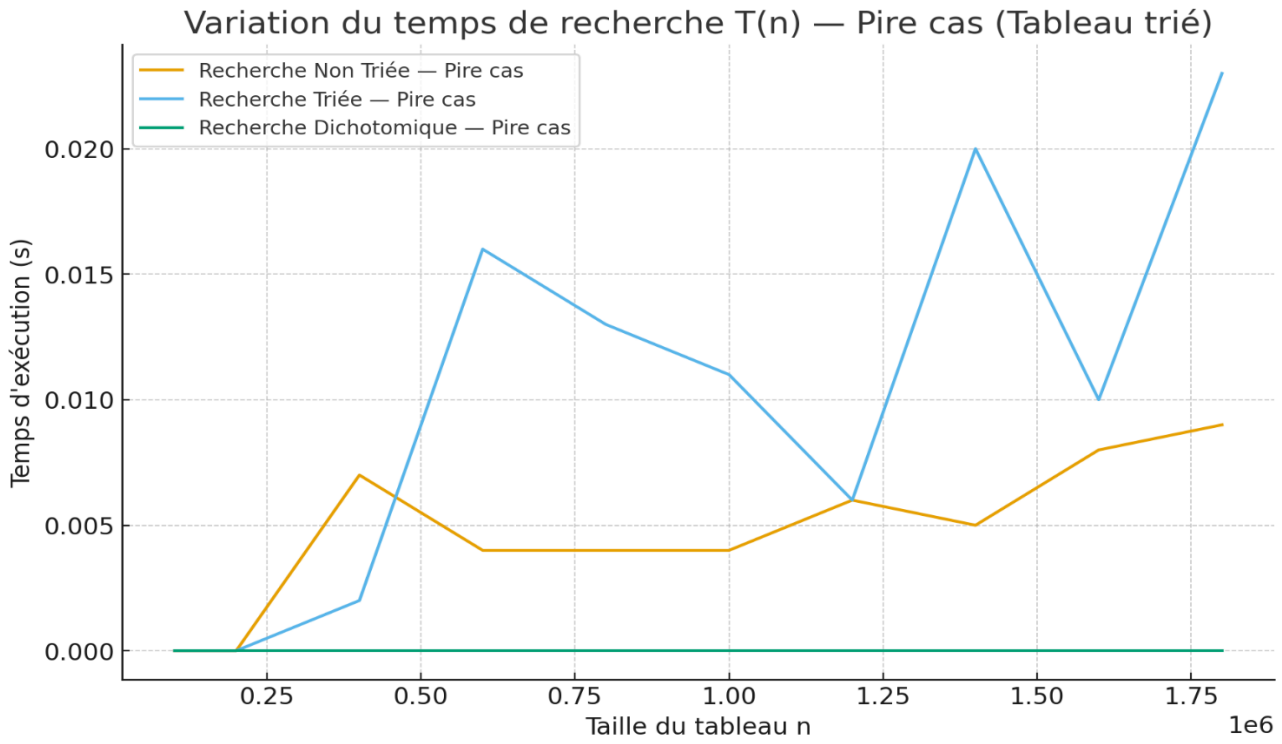
- **Pire cas :** O(log n)




3. Remplir le tableau suivant en donnant le temps de recherche1 de chacune des trois fonctions précédentes :

| *Méthode / n →* | *100000* | *200000* | *400000* | *600000* | *800000* | *1000000* | *1200000* | *1400000* | *1600000* | *1800000* |
|---|---|---|---|---|---|---|---|---|---|---|
| ***EletsNonTriés — Meilleur cas*** | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 |
| ***EletsNonTriés — Pire cas*** | 0.000000 | 0.000000 | 0.007000 | 0.004000 | 0.004000 | 0.004000 | 0.006000 | 0.005000 | 0.008000 | 0.009000 |
| ***EletsTriés — Meilleur cas*** | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 |
| ***EletsTriés — Pire cas*** | 0.000000 | 0.000000 | 0.002000 | 0.016000 | 0.013000 | 0.011000 | 0.006000 | 0.020000 | 0.010000 | 0.023000 |
| ***EletsTriesDicho — Meilleur cas*** | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 |
| ***EletsTriesDicho — Pire cas*** | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 |

4. Représenter dans un graphe les variations du temps d'exécution T(N) des 3 fonctions de recherches, quand le tableau est trié, dans le pire cas.




*5. Que constatez-vous ?*

En observant le graphe des temps d'exécution T(n) dans le pire cas pour les trois méthodes de recherche lorsque le tableau est trié, on constate les points suivants :

### 1) La recherche dichotomique est pratiquement instantanée

- Pour toutes les tailles n (100000 jusqu'à 1800000),
  **le temps mesuré est 0 seconde**.
- Cela s'explique par sa complexité **O(log n)**, extrêmement rapide.
- Même quand n augmente fortement, le temps ne change presque pas.

***Conclusion* :** La recherche dichotomique est de loin la plus efficace.

### 2) La recherche séquentielle dans un tableau trié est plus lente

- Elle affiche des temps entre **0.002 s et 0.023 s**.
- Son temps augmente globalement avec n (même si quelques petites variations existent selon la machine).
- C'est logique : en pire cas, il faut parcourir tout le tableau.

**Complexité : O(n)**

Donc le temps augmente proportionnellement à la taille du tableau.

### 3) La recherche séquentielle dans un tableau non trié est légèrement plus lente que la recherche triée

- Les temps obtenus sont entre **0.004 s et 0.009 s**.
- Elle semble un peu plus rapide que la recherche triée dans certains n, mais globalement :
  - **Elle reste linéaire O(n)**
  - Elle est plus lente que la recherche dichotomique

Cela est normal : dans le pire cas, il faut aussi parcourir tout le tableau.

## *Conclusion Générale*

1. **La recherche dichotomique est de loin la plus performante**, car elle divise le tableau en deux à chaque étape (O(log n)).
   → Son temps est presque constant et proche de zéro.
2. **Les recherches séquentielles (triée et non triée) sont beaucoup plus lentes**, car elles parcourent potentiellement tout le tableau (O(n)).
3. **Quand n augmente, les temps O(n) augmentent**, alors que ceux en **O(log n)** restent très faibles.

# A. Recherche du minimum et maxmimum

## 1. Approche naive :

*1.a. La fonction **MaxetMinA** de recherche du maximum et du minimum d'un tableau non trié de n élements :*

la foction implémentée parcourt sequentiellement le tableau en effectuant deux comparaisons pour chaque element :

```
void MaxEtMinA(int A[], int n, int* max, int* min){
    *max = A[0];
    *min = A[0];

    for(int i = 1; i < n; i++){
        if(A[i] > *max)
            *max = A[i];

        if(A[i] < *min)
            *min = A[i];
    }
}
```

Principe de fonctionnement :

i. Initialisation : on pose max = min = A[0]

ii. Pour chaque element A[i] (i allant de 1 à n-1) :

- On compare A[i] avec max actuel
- On compare A[i] avec min actuel

iii. On met à jour max et/ou min si necessaire

*1. b. Sa complixité théorique au pire cas :*

**Nombre de comparaisons :**

- Pour chaque element de A[1] à A[n-1], on effectue exactement 2 comparaisons
- Nombre total de comparaisons : **2(n-1) = 2n - 2** => **donc O(n).**

## 2. Algorithme plus efficace :

1. a. La fonction MaxEtMinB :

La fonction implémentée le principe décrit dans l'enoncé :

```c
void MaxEtMinB(int A[], int n, int* max, int* min){
    int start;

    if(n % 2 == 0){
        if(A[0] > A[1]){
            *max = A[0];
            *min = A[1];
        } else {
            *max = A[1];
            *min = A[0];
        }
        start = 2;
    } else {
        *max = *min = A[0];
        start = 1;
    }
```

```c
    for(int i = start; i < n - 1; i += 2){
        int grand, petit;

        // Comparaison
        if(A[i] > A[i+1]){
            grand = A[i];
            petit = A[i+1];
        } else {
            grand = A[i+1];
            petit = A[i];
        }

        // Mise à jour du maximum
        if(grand > *max)
            *max = grand;

        // Mise à jour du minimum
        if(petit < *min)
            *min = petit;
    }
}
```

Principe de fonctionnement :

Phase I - Initialisation :

- Si n est pair : on compare A[0] et A[1] pour initialiser max et min (1 comparaison)

- Si n est impair : on initialise max = min = A[0] (0 comparaison)

Phase II - Traitement par paires :

Pour chaque paire d'elements (A[i], A[i+1]) :

1. On compare A[i] et A[i+1] pour determiner le plus grand et le plus petit (1 comparaison)

2. On compare le plus grand avec max actuel (1 comparaison)

3. On compare le plus petit avec min actuel (1 comparaison)

Phase III - element restant :

- Si n est impair, le dernier element a deja ete considere lors de l'initialisation

2. *b. Complexité théorique au pire cas :*

**Cas ou n est pair :**

- Initialisation : 1 comparaison
- Nombre de paires : (n-2)/2 = n/2 - 1
- Comparaisons pour les paires : 3 × (n/2 - 1)
- Total : 1 + 3(n/2 - 1) = 1 + 3n/2 - 3 = 3n/2 - 2 comparaisons => O(n)

**Cas ou n est impair :**

- Initialisation : 0 comparaison
- Nombre de paires : (n-1)/2
- Comparaisons pour les paires : 3 × (n-1)/2
- Total : 3(n-1)/2 comparaisons => O(n)

## 3. Comparaison exprimentale avec compteurs :

| Taille | Comparaisons A | Comparaisons B |
|---|---|---|
| 100000 | 199998 | 149998 |
| 200000 | 399998 | 299998 |
| 400000 | 799998 | 599998 |
| 600000 | 1199998 | 899998 |
| 800000 | 1599998 | 1199998 |
| 1000000 | 1999998 | 1499998 |
| 1200000 | 2399998 | 1799998 |
| 1400000 | 2799998 | 2099998 |
| 1600000 | 3199998 | 2399998 |
| 1800000 | 3599998 | 2699998 |

**Verification Mathematique**

Pour n = 1 000 000 (pair) :

Methode A :

- Formule theorique : 2(n-1) = 2(1 000 000 - 1) = 1 999 998

- Resultat experimental : 1 999 998 => Correspondance parfaite

Methode B :

- Formule theorique : 3n/2 - 2 = 3(1 000 000) / 2 - 2 = 1 500 000 - 2 = 1 499 998

- Resultat experimental : 1 499 998 => Correspondance parfaite

## Conclusion :

i. Le ratio A/B est constant (≈ 4/3) quelle que soit la taille du tableau.

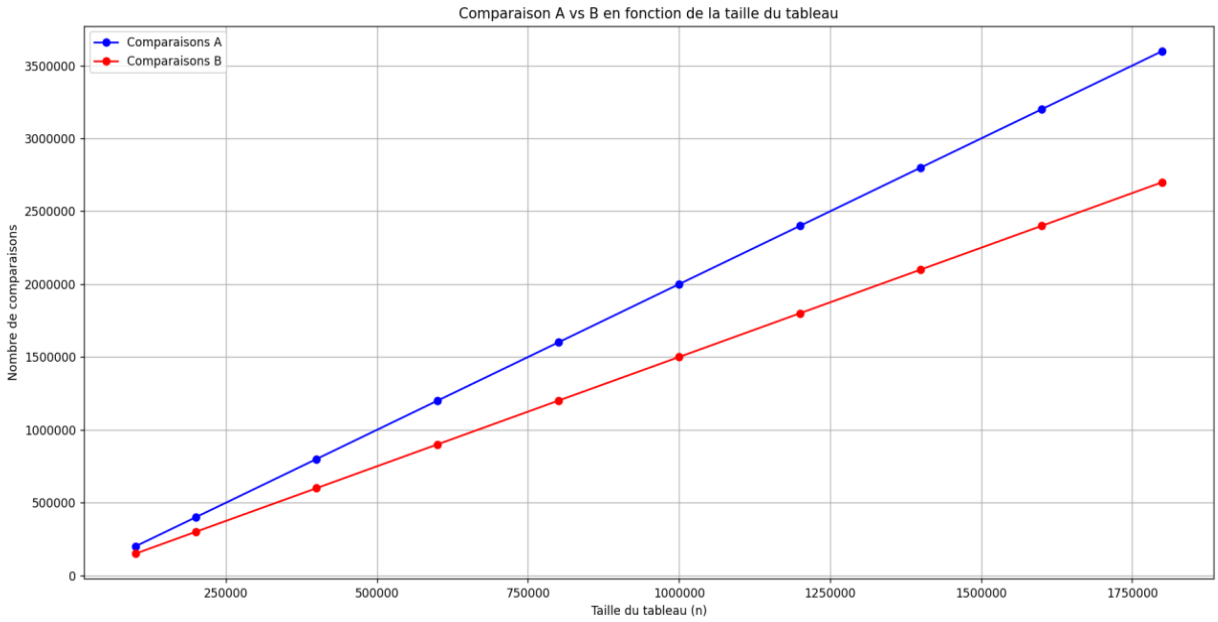
ii. La méthode A fait **4/3 fois plus** de comparaisons que B.

Autrement dit, B fait **3/4 = 75%** des comparaisons de A.

Donc B économise : **100% - 75% = 25%**

iii. Les formules theoriques sont validees experimentalement pour toutes les valeurs de n téstées.

## Graphe :




### Analyse graphique :

- Les deux courbes sont des droites parfaites, confirmant la complexite lineaire O(n).

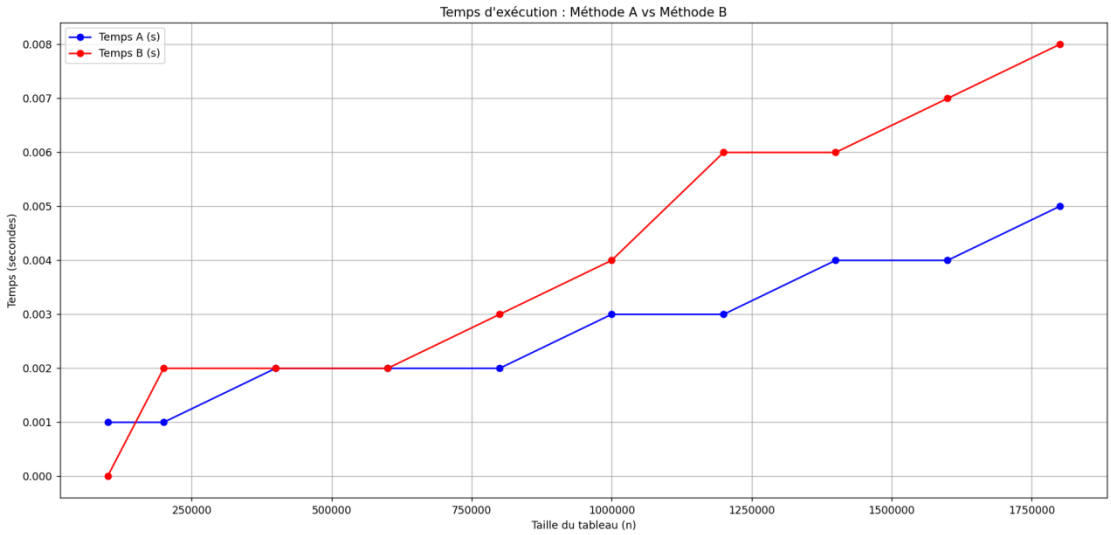
- La methode A (bleue) a une pente plus forte que la methode B (rouge).

## 4. Mesure et Comparaison des Complexites Temporelles :

### Resultas Temporels :

| Taille | Temps A (s) | Temps B (s)2 |
|---|---|---|
| 100000 | 0.001000 | 0.000000 |
| 200000 | 0.0010000 | 0.002000 |
| 400000 | 0.002000 | 0.002000 |
| 600000 | 0.002000 | 0.002000 |
| 800000 | 0.002000 | 0.003000 |
| 1000000 | 0.003000 | 0.004000 |
| 1200000 | 0.003000 | 0.006000 |
| 1400000 | 0.004000 | 0.006000 |
| 1600000 | 0.004000 | 0.007000 |
| 1800000 | 0.005000 | 0.008000 |

### Graphe :




### Analyse graphique :

- Les temps suivent une tendance globalement lineaire

- La methode B est generalement plus rapide que la methode A, Cependant, pour les petites valeurs de n, les mesures sont imprecises en raison de la faible resolution de l'horloge système.