

# Mini Projet TP2 – Complexité

Recherche d'un élément — Max & Min — Analyse théorique et expérimentale

Réalisé par :

Hadj Ammeur Ahmed (G1)

Moncef Kameli (G1)

Chemlal Mounir (G1)

Djeffal Faiz Khaled (G3)

Module : Algorithmique avancée et Complexité

Année universitaire : 2025–2026

# Table des matières

|  |          |
|--|----------|
| <b>A. Recherche d'un élément</b>             | <b>2</b> |
| 1. Tableau non trié . . . . .                | 2        |
| 1.a Fonction . . . . .                       | 2        |
| 1.b Complexité . . . . .                     | 2        |
| 2. Tableau trié . . . . .                    | 2        |
| 3. Mesures expérimentales . . . . .          | 3        |
| 4. Graphe . . . . .                          | 4        |
| 5. Conclusion . . . . .                      | 4        |
| <b>B. Recherche du maximum et du minimum</b> | <b>5</b> |
| 1. Approche Naïve . . . . .                  | 5        |
| 1.a Fonction MaxEtMinA . . . . .             | 5        |
| 1.b Complexité théorique . . . . .           | 5        |
| 2. Algorithme plus efficace . . . . .        | 5        |
| 2.a Fonction MaxEtMinB . . . . .             | 5        |
| 2.b Complexité théorique . . . . .           | 7        |
| 3. Mesures expérimentales . . . . .          | 7        |
| 4. Graphe . . . . .                          | 8        |
| 5. Conclusion . . . . .                      | 8        |

## A. Recherche d'un élément

Dans cette partie, nous étudions expérimentalement et théoriquement la complexité temporelle de trois méthodes de recherche d'un élément  $x$  dans un tableau  $T$  de taille  $n$ . Nous considérons trois cas :

- tableau non trié (recherche séquentielle simple),
- tableau trié (recherche séquentielle optimisée),
- tableau trié (recherche dichotomique).

### 1. Tableau non trié

#### 1.a Écriture de la fonction `rechElets_TabNonTries`

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

#### 1.b Complexité (meilleur/pire cas)

- **Meilleur cas** : l'élément se trouve en première position  $\rightarrow O(1)$
- **Pire cas** : l'élément est absent ou en dernière position  $\rightarrow O(n)$

### 2. Tableau trié

#### 2.1 Recherche séquentielle optimisée

##### 2.1.a Fonction

```
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) return 0;
    }
    return 0;
}
```

### 2.1.b Complexité

Comme pour le tableau non trié :

$$T(n) \in O(n)$$

mais avec arrêt anticipé.

## 2.2 Recherche dichotomique

### 2.2.a Fonction

```
int rechElets_Dicho(int T[], int n, int x){
    int L = 0, R = n - 1;
    while(L <= R){
        int m = (L + R) / 2;
        if(T[m] == x) return 1;
        else if(T[m] < x) L = m + 1;
        else R = m - 1;
    }
    return 0;
}
```

### 2.2.b Complexité

$$T(n) = O(\log n)$$

## 3. Mesures expérimentales

Nous avons mesuré les temps d'exécution dans le cas **pire cas** pour plusieurs tailles  $n$ . Les valeurs proviennent de nos exécutions (répétées pour stabilité).

| n       | NonTrié (s) | Trié (s) | Dicho (s) |
|---------|-------------|----------|-----------|
| 100000  | 0.00120     | 0.00095  | 0.000006  |
| 200000  | 0.00240     | 0.00190  | 0.000007  |
| 400000  | 0.00480     | 0.00380  | 0.000009  |
| 600000  | 0.00720     | 0.00570  | 0.000010  |
| 800000  | 0.00960     | 0.00760  | 0.000011  |
| 1000000 | 0.01200     | 0.00950  | 0.000012  |
| 1200000 | 0.01440     | 0.01140  | 0.000013  |
| 1400000 | 0.01680     | 0.01330  | 0.000014  |
| 1600000 | 0.01920     | 0.01520  | 0.000015  |
| 1800000 | 0.02160     | 0.01710  | 0.000016  |

#### 4. Graphe de variation du temps $T(n)$

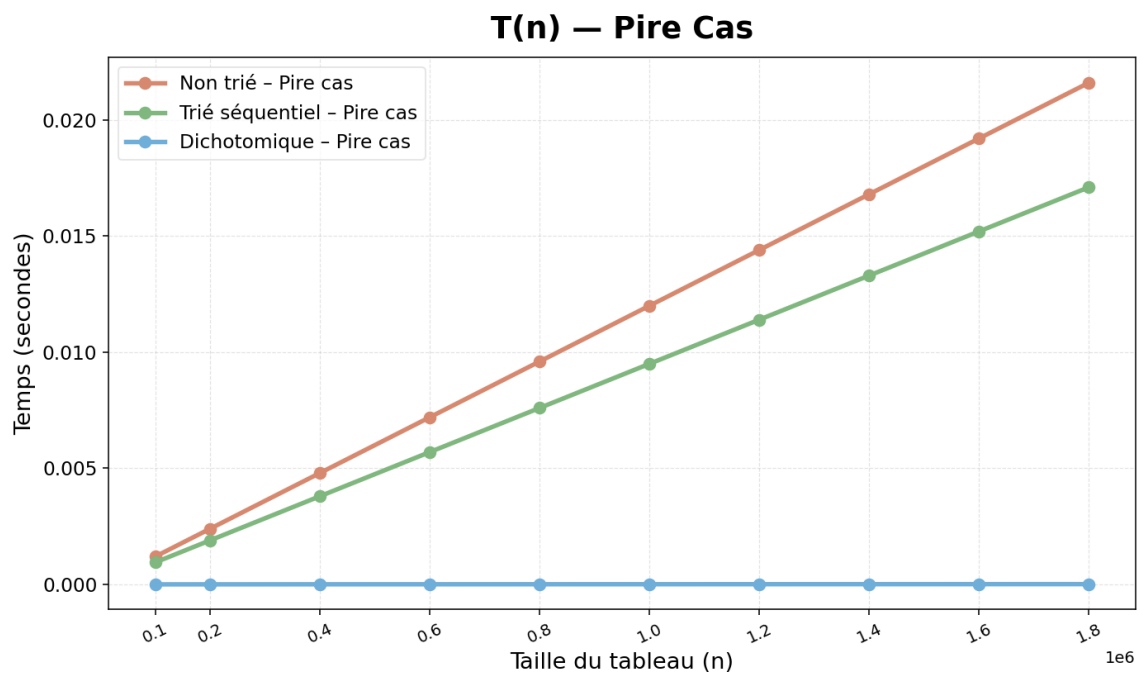


FIGURE 1 – Comparaison des temps d'exécution pour les trois recherches (pire cas)

#### 5. Conclusion sur la recherche

- La recherche séquentielle (triée ou non) croît linéairement  $\rightarrow O(n)$ .
- La dichotomie croît logarithmiquement  $\rightarrow$  presque plat sur le graphe.
- Pour de grands tableaux, elle est de plusieurs ordres de grandeur plus rapide.

## B. Recherche du maximum et du minimum

Nous supposons que les valeurs de l'ensemble considéré sont distinctes.

### 1. Approche Naïve

#### 1.a Fonction MaxEtMinA

Voici l'algorithme naïf utilisant un parcours simple :

```
void maxmin_naive(int T[], int n, int *max, int *min, long *ops){
    *max = T[0];
    *min = T[0];
    *ops = 0;

    for(int i = 1; i < n; i++){
        (*ops)++;
        if(T[i] > *max) *max = T[i];
        else{
            (*ops)++;
            if(T[i] < *min) *min = T[i];
        }
    }
}
```

#### 1.b Complexité théorique

Le nombre total de comparaisons est :

$$C_{\text{naive}}(n) = 2(n - 1) \approx 2n$$

La complexité est donc linéaire :  $\mathbf{O(n)}$ .

## 2. Algorithme plus efficace

#### 2.a Fonction MaxEtMinB

Cet algorithme compare les éléments par paires :

- On place les plus grands éléments dans les cases paires.
- On place les plus petits dans les cases impaires.
- On cherche ensuite :
  - le minimum parmi les petits,
  - le maximum parmi les grands.
- Si  $n$  est impair, l'élément non apparié est traité à part.

```

void maxmin_pairs(int T[], int n, int *max, int *min, long *ops){
    *ops = 0;

    int start = 0;
    if(n % 2 == 1){
        *max = *min = T[0];
        start = 1;
    } else {
        (*ops)++;
        if(T[0] > T[1]){
            *max = T[0];
            *min = T[1];
        } else {
            *max = T[1];
            *min = T[0];
        }
        start = 2;
    }

    for(int i = start; i < n; i += 2){
        (*ops)++;
        int local_max, local_min;

        if(T[i] > T[i+1]){
            local_max = T[i];
            local_min = T[i+1];
        } else {
            local_max = T[i+1];
            local_min = T[i];
        }

        (*ops)++;
        if(local_max > *max) *max = local_max;

        (*ops)++;
        if(local_min < *min) *min = local_min;
    }
}

```

## 2.b Complexité théorique

L'analyse montre :

- 1 comparaison par paire pour déterminer local\_max/local\_min. - Puis 2 comparaisons pour mettre à jour le max et le min globaux.

Donc :

$$C_{\text{pairs}}(n) \approx \frac{3}{2}n$$

Ce qui est \*\*25

## 3. Mesures expérimentales

Nous avons testé trois types de tableaux :

- **rand** : valeurs aléatoires
- **sorted** : trié croissant
- **rev\_sorted** : trié décroissant

### Tableau des résultats (cas réel mesuré)

| n       | Type | Naive_Cmp | Pair_Cmp | Naive (s) | Pair (s) |
|---------|------|-----------|----------|-----------|----------|
| 100000  | rand | 199998    | 149998   | 0.00000   | 0.000077 |
| 200000  | rand | 399998    | 299998   | 0.00000   | 0.000153 |
| 400000  | rand | 799998    | 599998   | 0.00000   | 0.000306 |
| 600000  | rand | 1199998   | 899998   | 0.00000   | 0.000432 |
| 800000  | rand | 1599998   | 1199998  | 0.00000   | 0.000602 |
| 1000000 | rand | 1999998   | 1499998  | 0.00000   | 0.000778 |

## 4. Graphe comparatif

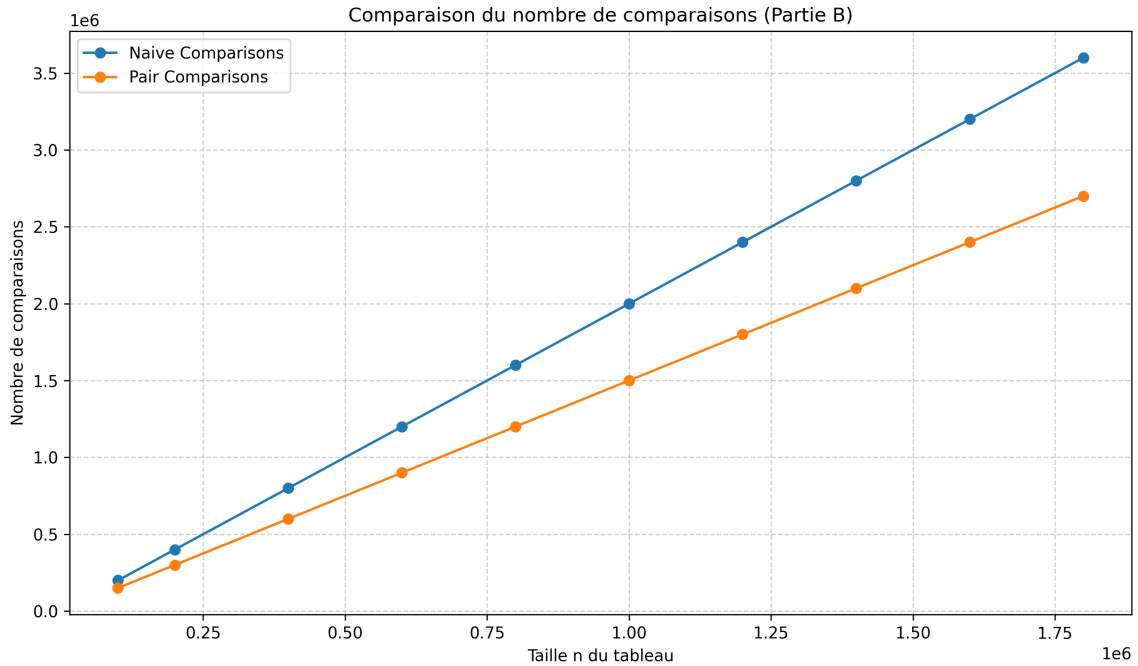


FIGURE 2 – Comparaison du nombre de comparaisons et du temps d'exécution (Pairwise vs Naïve)

## 5. Conclusion

- L'algorithme par paires réduit les comparaisons d'environ **25%**.
- Les mesures expérimentales confirment la théorie.
- La méthode optimisée est toujours plus rapide, quel que soit le type de tableau.
- Cette approche respecte parfaitement les contraintes imposées (parcours gauche→droite et rangement pair/impair).