Ecole Supérieure
d'Informatique et du Numérique
COLLEGE OF ENGINEERING & ARCHITECTURE

COMPTE RENDU

Algorithmique Avancée - TP2
3e année Cybersécurité - École Supérieure d'Informatique et du Numérique (ESIN)
Collège d'Ingénierie & d'Architecture (CIA)

Étudiant :	HATHOUTI Mohammed taha
Filière :	Cybersécurité
Année :	2025/2026
Enseignants :	M.BAKHOUYA
Date :	4 octobre 2025

1 Rappel des objectifs du TP

Ce TP fait suite au TP1 et approfondit l'étude des algorithmes de type "Diviser pour Régner". Nous avons implémenté et analysé deux exercices issus des travaux dirigés et travaux pratiques du chapitre 1 :

1.1 Exercice 2 (TP Chapitre 1) : Recherche binaire d'une valeur

L'objectif était d'implémenter la recherche binaire dans un tableau trié en utilisant :
— **Version itérative** : utilisation d'une boucle, complexité théorique $O(\log n)$
— **Version récursive** : appels récursifs, complexité théorique $O(\log n)$
Les expérimentations visaient à comparer les temps d'exécution réels avec les complexités théoriques respectives et à analyser les performances relatives de chaque approche.

1.2 Exercice 4 (TD Chapitre 1) - BONUS : Recherche Valeur Encadrée

L'objectif était d'implémenter un algorithme diviser-pour-régner permettant de trouver un élément $A[i]$ dans un tableau trié tel que $a \leq A[i] \leq b$ pour deux bornes données a et b.
Principe de l'algorithme :
— On divise le tableau en deux moitiés à partir de l'élément du milieu $A[m]$;
— Si $A[m] < a$ tous les éléments de la moitié gauche sont aussi $< a$, on cherche uniquement à droite ;
— Si $a \leq A[m] \leq b$: on a trouvé une valeur encadrée ;
— Si $b < A[m]$: tous les éléments de la moitié droite sont aussi $> b$, on cherche uniquement à gauche ;
— Complexité théorique : $O(\log n)$;
Pour comparaison, nous avons également implémenté une recherche linéaire naïve (complexité $O(n)$).

2 Implémentation

2.1 Structure du projet

Le projet est organisé en trois fichiers :
— `chercher.h` : déclarations des fonctions ;
— `chercher.c` : implémentation des algorithmes ;
— `main.c` : tests et mesures de performance ;

2.2 Algorithme récursif (Diviser pour Régner)

```
int chercher_recursif(int A[], int s, int f, int a, int b) {
    if (s == f) {
        if (A[s] >= a && A[s] <= b) {
            return s;
        } else {
            return -1;
```

```
        }
    }

    int m = (s + f) / 2;

    if (A[m] < a) {
        return chercher_recursif(A, m + 1, f, a, b);
    } else if (A[m] >= a && A[m] <= b) {
        return m;
    } else {
        return chercher_recursif(A, s, m, a, b);
    }
}
```

2.3 Algorithme linéaire (pour comparaison)

```
int chercher_lineaire(int A[], int n, int a, int b) {
    for (int i = 0; i < n; i++) {
        if (A[i] >= a && A[i] <= b) {
            return i;
        }
    }
    return -1;
}
```

3 Analyse expérimentale et résultats

3.1 Méthodologie

Les tests ont été effectués sur des tableaux triés de tailles croissantes (puissances de 2), allant de 1024 à 1 048 576 éléments. Pour chaque taille :

1. Génération d'un tableau d'entiers aléatoires
2. Tri du tableau avec `qsort()`
3. Définition des bornes : $a = A[n/3]$ et $b = A[2n/3]$
4. Mesure du temps d'exécution avec `clock()`
5. Validation des résultats

3.2 Résultats expérimentaux

3.2.1 Exercice 2 : Recherche binaire

Les deux implémentations de la recherche binaire ont été testées sur des tableaux triés de tailles croissantes :

Recherche binaire itérative :

— 1 048 576 éléments : 0.09 µs
— 2 097 152 éléments : 0.05 µs
— 4 194 304 éléments : 0.16 µs

— 8 388 608 éléments : 0.12 µs
— 16 777 216 éléments : 0.09 µs
— 33 554 432 éléments : 0.09 µs
— 67 108 864 éléments : 0.09 µs
— 134 217 728 éléments : 0.08 µs

Recherche binaire récursive :

— 1 048 576 éléments : 0.11 µs
— 2 097 152 éléments : 0.07 µs
— 4 194 304 éléments : 0.12 µs
— 8 388 608 éléments : 0.16 µs
— 16 777 216 éléments : 0.11 µs
— 33 554 432 éléments : 0.08 µs
— 67 108 864 éléments : 0.10 µs
— 134 217 728 éléments : 0.09 µs

Les résultats confirment la complexité $O(\log n)$ des deux méthodes. Les temps restent très stables (entre 0.05 et 0.16 µs) même quand on passe de 1 million à 134 millions d'éléments. Les deux versions donnent des performances quasiment identiques, ce qui correspond à la théorie.

3.2.2 Exercice 4 (BONUS) : Recherche Valeur Encadrée

Taille (n)	Linéaire (µs)	Récursif (µs)	Ratio
1 024	3.00	2.00	1.5×
2 048	3.00	1.00	3.0×
4 096	4.00	1.00	4.0×
8 192	8.00	1.00	8.0×
16 384	6.00	1.00	6.0×
32 768	10.00	1.00	10.0×
65 536	13.00	0.00	–
131 072	25.00	15.00	1.7×
262 144	48.00	1.00	48.0×
524 288	95.00	0.00	–
1 048 576	210.00	1.00	210.0×

TABLE 1 – Comparaison des temps d'exécution en microsecondes

3.3 Interprétation des résultats

3.3.1 Algorithme linéaire

Les résultats confirment la complexité $O(n)$:
— Croissance quasi-linéaire du temps d'exécution
— Pour 1 024 éléments : 3 µs
— Pour 1 048 576 éléments : 210 µs (facteur 70 pour une taille ×1024)

3.3.2 Algorithme récursif (Diviser pour Régner)

Les résultats valident la complexité $O(\log n)$:
— Temps pratiquement constant entre 0 et 2 µs

— Pas d'augmentation significative même pour 1 million d'éléments
— La fonction `clock()` n'a pas assez de précision pour mesurer ces temps ultra-courts

3.3.3 Comparaison

Le gain de performance devient spectaculaire sur de grandes données :
— Pour 1 048 576 éléments : gain de facteur $\times 210$
— L'écart se creuse proportionnellement à $n/\log n$
— L'algorithme diviser-pour-régner est clairement supérieur

3.4 Tests de validation

Quatre tests spécifiques ont été effectués :
Test 1 - Exemple du polycopié :
— Tableau : [3, 7, 8, 43, 556]
— Bornes : [40, 50]
— Résultat : indice 3 (valeur = 43)
Test 2 - Aucune valeur (toutes $< a$) :
— Tableau : [1, 2, 3, 4, 5]
— Bornes : [10, 20]
— Résultat : -1
Test 3 - Aucune valeur (toutes $> b$) :
— Tableau : [50, 60, 70, 80, 90]
— Bornes : [10, 20]
— Résultat : -1
Test 4 - Un seul élément :
— Tableau : [15]
— Bornes : [10, 20]
— Résultat : indice 0 (valeur = 15)
Tous les tests sont passés avec succès.

4 Analyse de complexité théorique

4.1 Algorithme linéaire

Cet algorithme parcourt le tableau élément par élément jusqu'à trouver une valeur encadrée :
— Meilleur cas : $O(1)$ (première valeur trouvée)
— Pire cas : $O(n)$ (aucune valeur, parcours complet)
— Cas moyen : $O(n/2) = O(n)$

4.2 Algorithme récursif (Diviser pour Régner)

À chaque appel récursif, l'espace de recherche est divisé par 2 :
Relation de récurrence :

$$T(n) = T(n/2) + O(1)$$

Application de la Méthode Générale (Master Theorem) :
— $a = 1$ (un seul appel récursif)
— $b = 2$ (division par 2)
— $f(n) = O(1)$
— Nous avons : $n^{\log_b a} = n^{\log_2 1} = n^0 = 1$
— Donc $f(n) = \Theta(n^{\log_b a}) \rightarrow$ Cas 2
— Conclusion : $T(n) = \Theta(\log n)$

4.3 Comparaison asymptotique

Algorithme	Complexité	Observation	Conformité
Rech. bin. itérative	$O(\log n)$	Temps constant	Oui
Rech. bin. récursive	$O(\log n)$	Temps constant	Oui
Rech. linéaire	$O(n)$	Croissance linéaire	Oui
Rech. encadrée	$O(\log n)$	Temps quasi-constant	Oui

TABLE 2 – Conformité théorie vs expérimentation

5 Conclusion

5.1 Points clés observés

Ce TP2 a permis de valider expérimentalement les complexités théoriques de plusieurs algorithmes de type "Diviser pour Régner" :

— **Recherche binaire** : les versions itérative et récursive offrent des performances identiques en $O(\log n)$;

— **Recherche encadrée (BONUS)** : l'algorithme D&C démontre sa supériorité avec un gain de performance spectaculaire ($\times 210$) par rapport à l'approche linéaire sur 1 million d'éléments ;

5.2 Enseignements

Ce TP2 complète le TP1 en approfondissant les algorithmes diviser-pour-régner. Nous avons pu constater que :

— L'analyse théorique (*MéthodeGénérale*) prédit correctement les performances réelles
— Les algorithmes en $O(\log n)$ sont extrêmement efficaces sur de grandes données
— La simplicité d'implémentation ne reflète pas toujours la performance : la recherche encadrée récursive est beaucoup plus rapide que l'algorithme linéaire pour les grandes tailles
— Il est crucial de choisir le bon algorithme en fonction de la taille des données traitées

L'exercice bonus sur la recherche encadrée a permis de vérifier concrètement les calculs de complexité vus en cours, renforçant ainsi la compréhension du lien entre théorie et pratique.