République Algérienne Démocratique et Populaire
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

Université des Sciences et de la Technologie Houari Boumédiène

Faculté d'Electronique et d'Informatique
Département Informatique

Master Systèmes Informatiques intelligents

Module : Conception et Complexité des Algorithmes

Rapport de projet de TP

Réalisé par :

Année universitaire : 2021 / 2022

Chapitre 1

Représentation d'une expression arithmétique en arbre binaire

1.1 Description de l'objectif de l'algorithme

Une expression arithmétique est une succession de caractères mathématiques notamment des nombres, des opérateurs mathématiques (+ addition, - soustraction, * multiplication, / division et * modulo) et des symboles impropres pour indiquer la priorité(( ) les parenthèses). Il faut noter cependant que les opérateurs mathématiques ne possèdent pas tous la même priorité ; La multiplication et la division sont plus prioritaires que l'addition et la soustraction. Dans le cas où la priorité de deux opérateurs est la même, celui le plus à gauche dans l'expression arithmétique devient plus prioritaire que celui à droite. En considérant ces contraintes, une des représentations les plus adaptées pour décrire une expression arithmétique est l'arbre binaire.

Un arbre binaire est une structure de données complexe, il est caractérisé par un élément racine qui contient à son tour un chemin vers un ou deux autres éléments appelés fils droit et fils gauche. Chaque élément intermédiaire de l'arbre par la suite a la même structure que la racine, jusqu'à arriver aux éléments terminaux qui ne possèdent pas de fils et qu'on nomme les feuilles de l'arbre.

Nous pouvons alors par extrapolation représenter chaque opération arithmétique par un élément de l'arbre, telle que l'élément de l'arbre contient l'opérateur, et les deux opérandes sont contenus dans les deux fils de l'élément. Pour qu'un arbre binaire puisse représenter la structure d'une expression arithmétique correctement, ce dernier doit respecter l'ordre et la priorité des opérations. Les opérations les moins prioritaires sont en haut de l'arbre car elles dépendent du résultat des opérations plus prioritaires qui elles, sont plus bas dans l'arbre binaire (voir Figure 1.1).

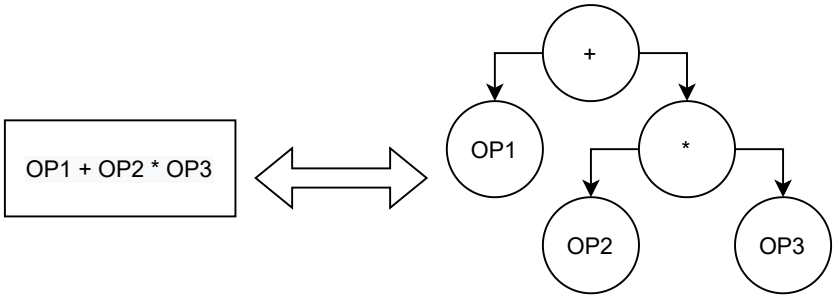

FIGURE 1.1 – Représentation d'une expression arithmétique en arbre binaire

1.2 Fonctionnement de l'algorithme

Le passage de l'expression arithmétique en arbre binaire passe par 2 étapes de traduction : l'analyse lexicale et puis la traduction dirigée par la syntaxe (voir Figure 1.2). L'expression arithmétique est d'abord lue du clavier sous forme de chaîne de caractères. Puis, un vecteur d'entités lexicales est généré à partir de cette chaîne. Enfin, on génère un arbre binaire à partir du vecteur d'entités lexicales.

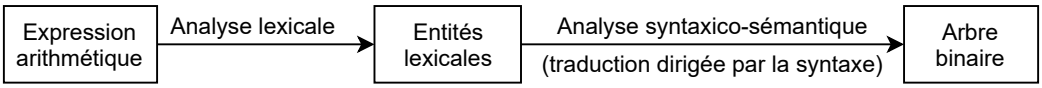

FIGURE 1.2 – Les étapes de la traduction

Les structures de données utilisées dans les algorithmes qui vont suivre sont les suivantes :

entité : représente une entité lexicale et contient les champs type de l'entité et la valeur.

noeud : représente un élément d'un arbre et contient une valeur et des pointeurs vers les fils droit et gauche du noeud.

arbre : représente un arbre binaire et contient un pointeur vers la racine de l'arbre.

1.2.1 Analyse lexicale

Dans cette partie, on extrait les entités lexicales de la chaîne de caractères contenant l'expression arithmétique. On parcourt la chaîne de caractères depuis le début, et on génère les entités lexicales selon les caractères lus.

Algorithme 1 : Analyse lexicale

Données : expression : chaîne de caractères
Résultat : entites : vecteur d'entités
Variables :
e : entité;
i, j : entier;
début
    Initialiser le vecteur entites à vide;
    pour $i \leftarrow 1$**à** *expression.taille*() **faire**
        si *expression[i] est un opérateur* **alors**
            `// Reconnaitre un opérateur`
            e.type←opérateur;
            e.operation←type_opération;
            entites.ajouter(e);
        sinon si *expression[i] est une parenthèse* **alors**
            `// Reconnaitre une parenthèse`
            e.type←parenthèse;
            e.parenthèse←type_parenthèse;
            entites.ajouter(e);
        sinon si *expression[i] est un chiffre ou un point* **alors**
            `// Reconnaitre un nombre`
            j←1;
            tant que $i + j \leq$ *expression.taille*() *et expression[i] est un chiffre ou un point* **faire** j←$j + 1$;
            e.type←nombre;
            e.nombre←en_nombre(expression.sous_chaine(i, j));
            entites.ajouter(e);
            i←$i + j - 1$;
        sinon
            Lever une exception; `// Erreur lexicale`
        fin si
    fin pour
fin

1.2.2 Analyse syntaxico-sémantique ou traduction dirigée par la syntaxe

Nous utilisons l'algorithme de descente récursive pour parcourir le vecteur d'entités et générer l'arbre binaire. Dans cet algorithme chaque MGP (membre de gauche de production) est associé à

une fonction dans le programme. L'algorithme commence en appelant une première fois l'axiome de la grammaire et se termine une fois que tout le vecteur est parcouru.

La grammaire utilisée est comme suit :

$$E \rightarrow T + \{T\}^* | T - \{T\}^*$$

$$T \rightarrow F * \{F\}^* | F / \{F\}^*$$

$$F \rightarrow nb | (E) | + F | - F$$

Algorithme 2 : Descente récursive

Données : expression : chaîne de caractères
Résultat : entites : vecteur d'entités
Variables :
tc : entier;
bt : arbre;
début
    ct←0;
    si $tc = entites.taille()$ **alors** Lever une exception `// Expression vide;`
    bt←e(entites, tc); `// Appel de la fonction e`
    retourner bt ;
fin

Cette fonction e traite les opérations d'addition et de soustraction qui sont moins prioritaires, et génère les noeuds correspondants. Le traitement des autres opérations est délégué à la fonction t qui va terminer son exécution avant la fonction e ; c-à-d. que les noeuds générés par les opérations plus prioritaires seront retournés à la fonction e qui va les ajouter dans les niveaux plus bas.

Fonction e(entites : vecteur d'entités, Entrée/Sortie tc : entier) : arbre

Variable :

bt, rt, lt : arbre;

e : entité;

début

bt←t(entites, tc); `// Appel de la fonction t`

tant que *tc* ≤ *entites.taille*() *et entites[tc] est un opérateur d'addition ou de soustraction* **faire**

e←entites[tc];

lt←bt;

tc←tc + 1;

si *tc* > *entites.taille*() **alors** Lever une exception `// Erreur syntaxique`;

rt←t(entites, tc); `// Appel de la fonction t`

bt.racine←e;

bt.fils_gauche←lt;

bt.fils_droit←rt;

fin tq

retourner *bt*;

fin

Cette fonction t traite les opérations de multiplication, de division et de modulo. Elle utilise la fonction f pour générer les noeuds des nombres et des sous-expressions plus prioritaires comme les parenthèses.

```
Fonction t(entites : vecteur d'entités, Entrée/Sortie tc : entier) : arbre
  Variable :
  bt, rt, lt : arbre;
  e : entité;
  début
    bt←f(entites, tc);                                   // Appel de la fonction f
    tant que tc ≤ entites.taille() et entites[tc] est un opérateur de multiplication ou de
     division ou de modulo faire
      e←entites[tc];
      lt←bt;
      tc←tc + 1;
      si tc > entites.taille() alors Lever une exception      // Erreur syntaxique;
      rt←f(entites, tc);                                 // Appel de la fonction f
      bt.racine←e;
      bt.fils_gauche←lt;
      bt.fils_droit←rt;
    fin tq
    retourner bt;
  fin
```

Cette fonction f génère les noeuds des nombres et des opérateurs unaires, elle repasse le contrôle à la fonction e en cas d'utilisation de parenthèses pour traiter la sous-expression contenue dedans.

```
Fonction f(entites : vecteur d'entités, Entrée/Sortie tc : entier) : arbre
  Variable :
  bt, rt, lt : arbre;
  e, gauche, droit : entité;
  début
    si entites[tc] est une parenthèse ouvrante alors
      tc←tc + 1;
      si tc > entites.taille() alors Lever une exception          // Erreur syntaxique;
      bt←e(entites, tc);                                        // Appel de la fonction e
      si tc ≤ entites.taille() et entites[tc] est une parenthèse fermante alors
        tc←tc + 1;
      sinon
        Lever une exception                                      // Erreur Syntaxique
      fin si
    sinon si entites[tc] est un opérateur d'addition ou de soustraction alors
      // Cas d'un opérateur unaire
      e←entites[ct];
      gauche.type←type_nombre;
      gauche.nombre←0;
      tc←tc + 1;
      si tc > entites.taille() alors Lever une exception          // Erreur syntaxique;
      rt←f(entites, tc);                                        // Appel de la fonction f
      lt.racine←gauche;
      bt.racine←e;
      bt.fils_gauche←lt;
      bt.fils_droit←rt;
    sinon
      // Cas d'un nombre (feuille de l'arbre)
      e←entites[tc];
      bt.racine←e;
      tc←tc + 1;
    fin si
    retourner bt;
  fin
```

1.3 Calcul de complexité

1.3.1 Complexité temporelle

La complexité de l'analyse lexicale est toujours égale à la longueur de la chaîne dans tous les cas c-à-d. : $\mathcal{O}(n)$ tel que n est la longueur de l'expression en de caractères.

La complexité de l'analyse syntaxique est quant à elle égale à la longueur du vecteur d'entités lexicales, car il est parcouru une seule fois, c-à-d. : $\mathcal{O}(n')$ tel que n' est la longueur du vecteur d'entités.

La complexité temporelle de l'algorithme devient alors : $\mathcal{O}(n + n')$.

1.3.2 Complexité spatiale

L'expression arithmétique est d'abord stockée dans une chaîne de caractères de longueur n, puis dans un vecteur d'entités de longueur n', puis dans un arbre binaire qui contient lui aussi n' éléments.

Sachant qu'une entité contient le type de l'entité et la valeur en elle-même, cependant la valeur est stockée comme un nombre réel qui prend 8 octets comparé à la représentation en chaîne de caractères où chaque caractère est sous 1 octet.

Donc la taille d'un élément d'une chaîne de caractères est 1 unité (octet), la taille d'une entité est 9 unités, et la taille d'un élément d'un arbre est égale à $9 + 4 * 2$ unités (la taille d'un pointeur est égale à 4 unités) donc 17 unités.

La complexité spatiale est égale à la somme des trois complexités (taille de la chaîne + taille du vecteur + taille de l'arbre) : $n * 1 + n' * 9 + n' * 17 = n + n' * 26 \approx \mathcal{O}(n + n')$

1.4 Expérimentation

Le tableau suivant représente les temps d'exécution en nanoseconde de l'algorithme selon la variation de la taille de l'expression arithmétique en d'opérande.

N	10	50	100	500	1000	5000	10000	100000	1000000
t1(ns)	177584	437769	381575	418252	686466	3111420	6786260	60986100	642610000
t2(ns)	101653	193648	339100	354777	2322770	3307840	20424500	59967200	639145000
t3(ns)	31972	47574	216601	354751	2115200	3375350	7450650	60956700	639422000
t4(ns)	100708	149167	259485	337421	820084	3283810	19698800	61631500	641354000
t5(ns)	24343	140434	95821	345987	683119	5158840	7575520	61750100	640897000
t6(ns)	30901	161277	337395	807634	671850	3292160	21818400	61473700	641686000
t7(ns)	52941	140159	288308	329206	693785	4360970	8418610	60870500	667895000
t8(ns)	69185	163825	242197	520649	2050330	5460620	22419800	63008000	639987000
t9(ns)	69510	97011	93270	356979	692529	5279490	6420450	61725500	640873000
t10(ns)	83819	142928	92484	1071680	803473	13108600	6558390	60088300	639895000
Moyenne(ns)	74262	167379	234624	489734	1153961	4973910	12757138	61245760	643376400

La figure suivante (voir Figure 1.3) représente l'évolution du temps d'exécution selon la longueur de l'expression arithmétique en d'opérande.

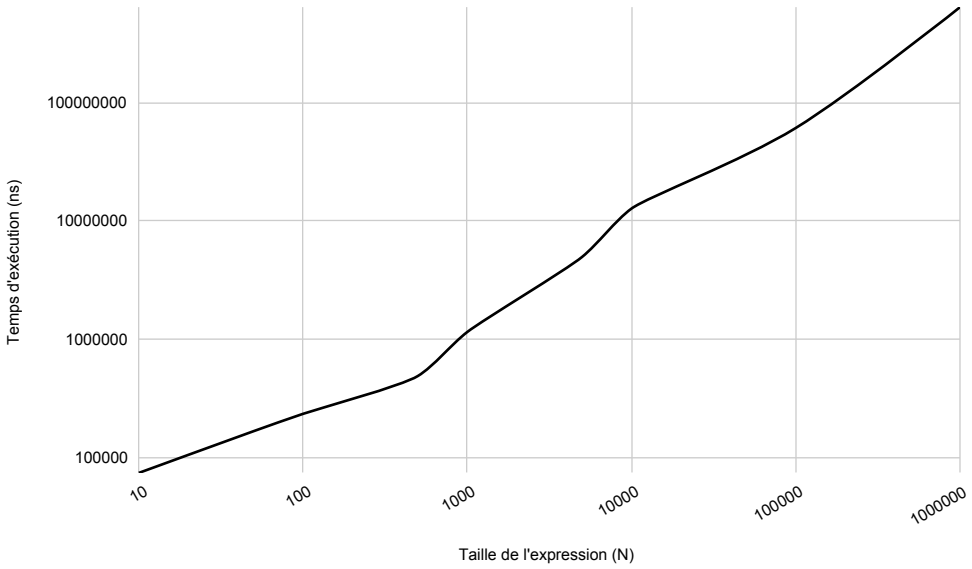

FIGURE 1.3 – Temps d'exécution du programme selon la longueur de l'expression en d'opérandes

Depuis le graphe, la courbe est sous forme d'une droite, on observe que le temps d'exécution évolue de manière linéaire avec l'augmentation de la taille du problème, ce qui correspond bien à la complexité théorique calculée auparavant.

1.5 Conclusion

L'algorithme de la descente récursive propose une complexité optimale pour la représentation de l'expression arithmétique en arbre binaire, égale à la longueur de l'expression arithmétique, car

il parcourt l'expression un nombre minimal de fois (une seule fois). Nous avons bien vu pendant les expériences que l'évolution du temps d'exécution d'une telle complexité est relativement contrôlée et suit une courbe droite et linéaire.

Chapitre 2

Recherche dichotomique d'un élément dans un tableau

2.1 Description de l'objectif de l'algorithme

En informatique, un tableau est une structure de données représentant une séquence finie d'éléments défini par un index représentant sa position au sein du tableau nous permettant d'y accéder. C'est un type de conteneur que l'on retrouve dans un grand nombre de langages de programmation et est l'un des plus utilisés du de sa simplicité.

Les données du tableau étant accessible individuellement il est nécessaire de faire une recherche lorsque l'on souhaite accéder a une valeur spécifique du tableau cependant lorsque la taille de la structure est grande il devient difficile d'y accéder efficacement, c'est pour cela que de nombreux algorithmes ont été conçu afin d'optimiser cette tache est l'un de ses algorithmes les plus performants est celui de la recherche dichotomique.

La recherche dichotomique ou recherche par dichotomie, est un algorithme de recherche pour trouver la position d'un élément dans un tableau. La seule condition a son application étant que le tableau soit trie, il est utilisable dans de nombreuses problématiques. Son principe consiste comparer l'élément avec la valeur de la case au milieu du tableau, si les valeurs sont égales, on met fin à l'exécution, sinon si la valeur recherchée est inférieur à la valeur de la case au milieu, on recommence dans la moitié du tableau contenant les valeurs plus petites que celle située au milieu du tableau et dans le cas contraire on prend la moitie contenant les valeurs supérieures, et ceci jusqu'à avoir trouvé la valeur souhaite ou avoir un sous tableau n'ayant qu'une seule valeur empêchant de continuer la recherche dans le cas ou la valeur n'existe pas dans le tableau. (voir Figure 2.1).

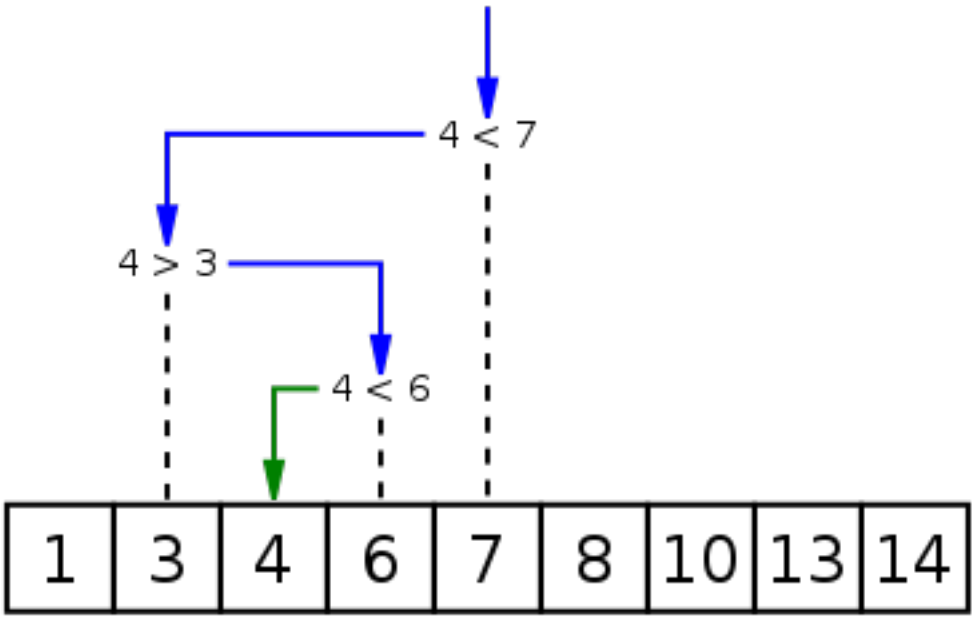

FIGURE 2.1 – Exemple graphique d'une recherche dichotomique

2.2 Fonctionnement de l'algorithme

Il nous est possible de distinguer 2 étapes distinctes et essentiels au bon déroulement de cet algorithme la première étant le trie du tableau, car comme précisé précédemment la recherche par dichotomie nécessite que le tableau soit trié et ne pouvant pas garantir de recevoir un tableau trié en entrée il est nécessaire de le trier au préalable avant de commencer le recherche afin de pouvoir le plus de cas que possible, par la suite la seconde étape consiste à appliquer la recherche dichotomique.

2.2.1 Tri du tableau

Dans cette partie, on applique un tri ascendant sur tableau et pour cela nous appliquons l'algorithme de Tri Fusion.

Le tri par fusion aussi appeler tri dichotomique est un exemple classique d'algorithme de division pour régner. L'opération principale de l'algorithme est la fusion, qui consiste à réunir deux listes triées en une seule. L'efficacité de l'algorithme vient du fait que deux listes triées peuvent être fusionnées en temps linéaire (voir Figure 2.2). On peut résumer son fonctionnement en deux étapes :

1. Divisez la liste non triée en sous-listes jusqu'à ce qu'il y ait N sous-listes avec un élément dans chacune (N est le nombre d'éléments dans la liste non triée).
2. Fusionnez les sous-listes deux à la fois pour produire une sous-liste triée, répétez cette opération jusqu'à ce que tous les éléments soient inclus dans une seule liste.

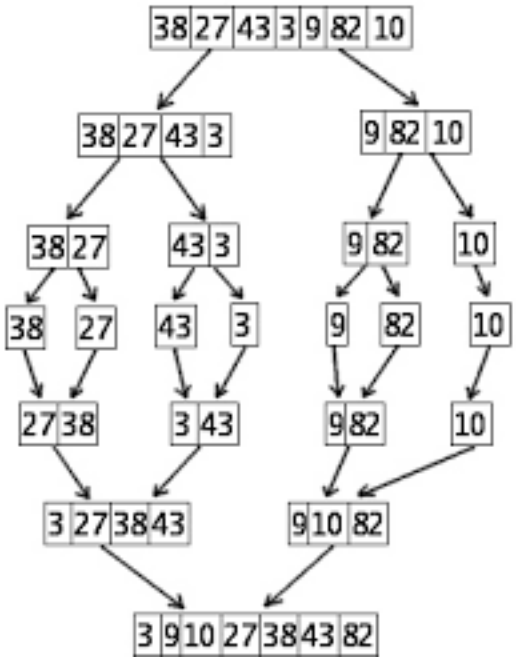

FIGURE 2.2 – Exemple graphique d'un tri dichotomique

Nous pouvons le représenter via le pseudo code suivant :

Fonction Fusion(Entrée : tab : tableau d'entier ; droite, gauche, milieu : entier ;)

Variables :

TabG, TabD : tableau d'entier;

SousTab1, SousTab2, SousTab1Index, SousTab2Index, SousTabFusionIndex : entier;

début

$SousTab1 \leftarrow milieu - gauche + 1$;

$SousTab2 \leftarrow droite - milieu$;

pour $i \leftarrow 1$ **à** $SousTab1$ **faire**

$tabG[i] \leftarrow tab[gauche + i]$;

fin pour

pour $j \leftarrow 1$ **à** $SousTab2$ **faire**

$tabG[j] \leftarrow tab[mid + 1 + j]$;

fin pour

$SousTab1 \leftarrow 0$;

$SousTab2 \leftarrow 0$;

$SousTabFusionIndex \leftarrow gauche$;

tant que $SousTab1Index < SousTab1$ *et* $SousTab2Index < SousTab2$ **faire**

si *tabG[SousTab1Index] <= tabD[SousTab2Index]* **alors**

$tab[SousTabFusionIndex] \leftarrow tabG[SousTab2Index]$;

$SousTab1Index + +$;

fin si

sinon

$tab[SousTabFusionIndex] \leftarrow tabD[SousTab2Index]$;

$SousTab2Index + +$;

fin si

$SousTabFusionIndex + +$;

fin tq

tant que $SousTab1Index < SousTab1$ **faire**

$tab[SousTabFusionIndex] \leftarrow tabG[SousTab1Index]$;

$SousTab1Index + +$;

$SousTabFusionIndex + +$;

fin tq

tant que $SousTab2Index < SousTab2$ **faire**

$tab[SousTabFusionIndex] \leftarrow tabG[SousTab2Index]$;

$SousTab1Index + +$;

$SousTabFusionIndex + +$;

fin tq

fin

Fonction TriFusion(Entrée : tab : tableau d'entier ; debut, fin : entier ;)

Variables :

milieu : entier;

début

```
// On divive le tableau en 2 de manière recursive puis on les tri avant
   de les fusionner
```

si $debut >= fin$ **alors**

retour;

fin si

```
// On calcule l'index du milieu du tableau
```

$milieu \leftarrow debut + (fin - debut)/2$;

TriFusion(tab, debut, milieu);

TriFusion(tab, milieu + 1, fin);

```
// On utilise la fonction fusion pour trier puis fusioner les sous
   tableaux en un seul tableau trié
```

Fusion(tab, debut, mid, fin);

fin

Afin d'optimiser le déroulement du tri, nous utilisons 2 fonctions distinctes. Une fonction TriFusion qui divise le tableau récursivement et une fonction Fusion qui elle trie les sous tableaux avant de les fusionner à nouveau

2.2.2 Recherche dichotomique

La recherche dichotomique consiste à rechercher dans un tableau trié en divisant l'intervalle de recherche en deux. On commence par un intervalle couvrant tout le tableau. Si la valeur de la clé de recherche est inférieure à l'élément situé au milieu de l'intervalle, on limite l'intervalle à la moitié inférieure. Sinon, le réduire à la moitié supérieure. On vérifie à plusieurs reprises jusqu'à ce que la valeur soit trouvée ou que l'intervalle soit vide.

Nous pouvons le représenter via le pseudo code suivant :

```
Fonction dichotomie(Entrée : tab : tableau d'entier ; n, r : entier ; Sortie : pos)
  Variables :
  debut, fin, milieu : entier;
  début
    // si la valeur recherchée est superieur ou inferieur au bornes du
       tableau on retourne -1 si elle est égale a une des bornes on retourn
       leur positions
    si r < tab[0] ou r > tab[n − 1] = alors
      pos ← −1;
      retour pos;
    fin si
    si tab[0] == r alors
      pos ← 0;
      retour pos;
    fin si
    si tab[n − 1] == r alors
      pos ← n − 1;
      retour pos;
    fin si
    tant que fin >= debut faire
      // Calcul de la position de l'élément du milieu
      i ← (debut + fin)/2;
      // Si l'élément du milieu est l'élément recherché on retourne sa
         position
      si tab[i] == r alors
        pos ← i;
        retour pos;
      fin si
      // Si la valeur recherchée est plus petite que la valeur du l'élément
         du milieu Alors on regarde le sous-tableau de gauche
      tab[i] > r fin = i − 1;
      // sinon on regarde le sous-tableau de droite
      sinon
        debut = i + 1;
      fin si
    fin tq
  fin
```

2.3 Calcul de complexité

2.3.1 Complexité temporelle

La complexité du tri dichotomique est toujours égale à : $\mathcal{O}(n\log(n)$ que ce soit dans le cas meilleur ou le cas tel que n est la longueur du tableau.

La complexité de la recherche dichotomique est quant à elle égale à $\mathcal{O}(\log(n')$ tel que n' est la longueur du tableau.

La complexité temporelle de l'algorithme devient alors : $\mathcal{O}(n\log(n)+\log(n')) = O(\log(n^n+n'))$.

Le tableau suivant représente les temps d'exécution théorique en nanoseconde de l'algorithme selon la variation de la taille du tableau

N	10	50	100	500	1000	5000	10000	100000	1000000	10000000
t(ns)	11	87	202	1352	3003	18499	40004	500005	6000006	70000007

La figure suivante (voir Figure 2.3) représente l'évolution du temps d'exécution théorique selon la longueur du tableau.

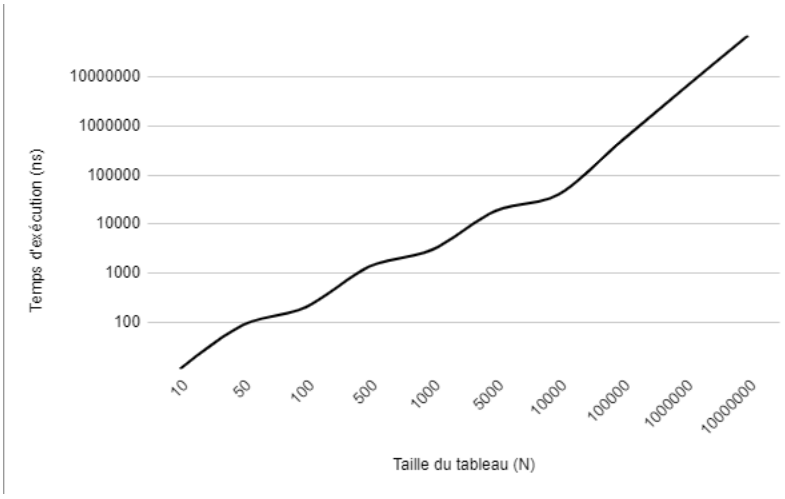

FIGURE 2.3 – Temps d'exécution théorique du programme selon la longueur du tableau

Depuis le graphe, on observe que le temps d'exécution évolue de manière linéarithmique avec l'augmentation de la taille du problème.

2.3.2 Complexité spatiale

L'expression arithmétique est d'abord stockée dans une chaîne de caractères de longueur n, puis dans un vecteur d'entités de longueur n', puis dans un arbre binaire qui contient lui aussi n' éléments.

L'unique structure de données utilisée est un tableau d'entier a n éléments.

La taille d'un entier étant de 2 octets la complexité spatiale est égale au produit de la taille du tableau et de la taille d'un entier : $n * 2 \approx \mathcal{O}(n)$

2.4 Expérimentation

Le tableau suivant représente les temps d'exécution expérimentale en nanoseconde de l'algorithme selon la variation de la taille du tableau.

N	10	50	100	500	1000	5000	10000	100000	1000000	10000000
t1(ns)	2164	28704	92760	169733	341583	1047560	4598740	35465600	5626090000	37018600000
t2(ns)	6674	35401	75401	251223	600691	1642120	3621170	34767200	3774520000	33572800000
t3(ns)	5654	33403	53403	275397	771243	1225350	2874540	40542400	2998740000	36542700000
t4(ns)	4233	42553	82553	251567	477149	988353	4714569	37218300	3411720000	36325400000
t5(ns)	5904	58861	58861	163133	348047	1253250	3148226	37992800	3845760000	29451700000
t6(ns)	5645	20634	70634	252684	801493	1991560	2132380	40874100	298370000	401745900000
t7(ns)	3820	43164	64307	251082	481145	1035130	3258380	37476600	3220570000	36945100000
t8(ns)	5641	32783	57953	283887	428472	2290770	2811460	28514100	3859450000	28416500000
t9(ns)	7242	47059	65421	258297	429683	1988180	3223750	3589600	3042580000	35473800000
t10(ns)	5242	31262	41654	233613	530896	1041460	3706200	45122700	1687400000	33447800000
Moyenne(ns)	5221	37382	66294	239061	521040	1271437	3429518	37552644	3176520000	70894030000

La figure suivante (voir Figure 2.4) représente l'évolution du temps d'exécution selon la longueur du tableau.

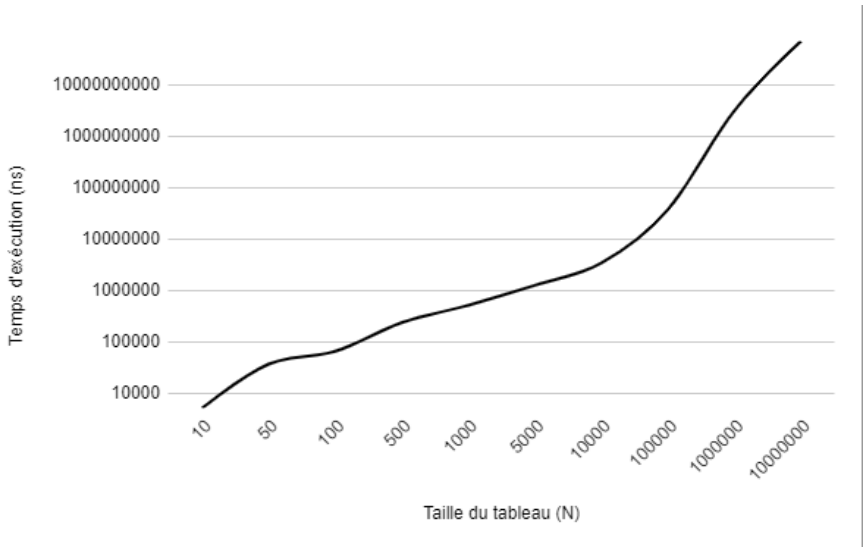

FIGURE 2.4 – Temps d'exécution du programme selon la longueur du tableau

Depuis le graphe, on observe que le temps d'exécution évolue de manière linéarithmique avec l'augmentation de la taille du problème, ce qui correspond bien à la complexité théorique et d'exécution théorique calculée auparavant.

2.5 Conclusion

L'algorithme de la recherche dichotomique propose une complexité optimale pour la recherche dans un tableau, cependant sa dépendance d'une fonction de tri afin de pouvoir fonctionner Quel que soit le tableau donné impacte négativement sa complexité, dans notre l'utilisation d'une fonction de tri dichotomique aura fait passer notre algorithme d'une complexité logarithmique a une complexité linéarithmique. Nous pouvons en conclure que la recherche par dichotomie est effectivement une bonne option, mais qu'afin de pouvoir l'utiliser à son plein potentiel il est nécessaire de l'utiliser sur des tableaux préalablement triés ou accompagné d'une fonction de tri ayant une complexité égale ou meilleure à elle.