Devoir maison Algo et complexity

étudiant: Bouzara Zakaria
matricule: 212138069681

I. Considérant un parcours séquentiel

```
fonction trouver_cle(E: A [n] entier, clé entier, S: entier) { // n est la taille du ta
    pour i=0, i < n , i++ faire
        si A[i] == clé alors
            retourner i
        fsi
    fait
}
```

La complexité de cet algorithme:

2- le pire des cas est le cas quand l'élément que nous recherchons est le dernier élément du tableau A avec complexité $O(n)$
3- le meilleur des cas est quand la clé est le premier élément du tableau (c'est le cas aussi quand n = 1) avec complexité $O(1)$

4. le même algorithme avec récursivité:

```
fonction trouver_clé(E: A [n] entier, clé entier , k entier ,S: entier){
    si A[1] == clé alors
        retourner k
    sinon
        retourner trouver_clé(A[2:n], clé, k+1)
fin
trouver_clé(A, clé, 1) // il faut mettre k=1 pour cet algorithme
```

5. en cas qu'on n'est pas sûr que clé appartienne au tableau

on peut mais le résultat de l'algorithme = -1 si la clé n'appartient pas au tableau A pour l'algorithme itératif on peut ajouter un `retourne` à l'extérieur de la boucle pour,

```
fonction trouver(E: A [n] entier, clé entier, S: entier) { // n est la taille du table
```

```
    pour i=0, i < n , i++ faire
        si A[i] == clé alors
            retourner i
        fsi
    fait
    retourner -1
}
```

ça veut dire si la boucle parcourt le tableau A sans retourner elle va retourner -1

pour l'algorithme récursif on peut ajouter une condition si la taille `n` du tableau A = 1 et le dernier élément A n'est pas équivalent à la clé la fonction va retourner -1

```
fonction trouver_clé(E: A [n] entier, clé entier , k entier ,S: entier){
    si A[1] == clé alors
        retourner k
    sinon si n == 1 alors
        retourner -1
    sinon
        retourner trouver_clé(A[2:n], clé, k+1)
}
```

II. Considérant un parcours dichotomique

```
fonction trouver_clé(E: A [n] entier, clé entier, S: entier) {
    droite = n
    gauche = 0
    mid = n // 2
    tant que droite >= gauche faire
        si clé = A[mid] alors
            retourner mid
        sinon si clé > A[mid] alors
            gauche = mid + 1
        sinon
            droite = mid - 1
        mid = ( droite + gauche ) // 2
    fait
}
```

La complexité de cet algorithme: 2- le pire des cas est le cas quand l'élément que nous recherchons est à l'une des 4 extrémités (premier élément, dernier élément ou élément adjacent du milieu) dans ce cas la complexité est de $O(log_2(n))$ 3- le meilleur des cas est quand la clé est l'élément médian du tableau avec complexité $O(1)$

4. le même algorithme avec récursivité:

```
fonction trouver_clé(E: A [n] entier, clé, k entier, S: entier) {
    // k correspondant à l'index logique du premier élément du tableau

    mid = n // 2
    si clé = A[mid] alors
        retourner ( k + mid - 1 )
    sinon si clé > A[mid] alors
        retourner trouver_clé(A[mid+1:], clé, k + mid)
    sinon
        retourner trouver_clé(A[:mid-1], clé, k)
    finsi
}
```

5. en cas qu'on n'est pas sûr que clé appartienne au tableau

- pour l'algorithme itératif on ajoute un retourne après la boucle tant que

```
tant que droite >= gauche faire
    si clé = A[mid] alors
        retourner mid
    sinon si clé > A[mid] alors
        gauche = mid + 1
    sinon
        droite = mid - 1
    mid = ( droite + gauche ) // 2
fait
retourner -1
```

- pour l'algorithme récursif on ajoute une autre condition d'arrêt quand la taille `n` de A est = 1 et l'élément du tableau A n'est pas la clé que nous cherchons:

```
si clé = A[mid] alors
    retourner ( k + mid - 1 )
sinon si n == 1 alors // le tableau contient un seul élément qui n'est pas égal à 
    retourner ( -1 )
sinon si clé > A[mid] alors
    retourner ( trouver_clé(A[mid+1:], clé, k + mid) )
sinon
    retourner (trouver_clé(A[:mid-1], clé, k))
finsi
```

Voici les corrections orthographiques apportées sans changer les phrases :

Considérons un arbre binaire

I. Parcours séquentiel dans l’arbre binaire logique

1) Algorithme itératif

```
fonction parcours_sequentiel(E: arbre [n] entier, cle entier, S: entier) {
    pour i de 0 à n faire
        si arbre[i] == cle alors
            retourner i
    fait
fin
```

2) Pire des cas et complexité

Le pire des cas correspond à la situation où la clé recherchée est le dernier élément du tableau. Dans ce cas, la complexité de l'algorithme est (O(n)).

3) Meilleur des cas et complexité

Le meilleur des cas correspond à la situation où la clé recherchée est le premier élément du tableau. Dans ce cas, la complexité de l'algorithme est (O(1)).

4) Algorithme récursif

Voici le même algorithme sous forme récursive :

```
fonction parcours_recursif(E: arbre [n] entier, cle, k entier, S: entier) {
    si arbre[k] == cle alors
        retourner k
    finsi
    retourner parcours_recursif(arbre, cle, k + 1)
}
```

5) Modification des versions si la clé n'appartient pas au tableau

Pour l'algorithme itératif, on retourne simplement -1 à la fin de la boucle si la clé n'est pas trouvée :

```
fonction parcours_sequentiel(E: arbre [n] entier, cle entier, S: entier) {
```

```
    pour i de 0 à n faire
        si arbre[i] == cle alors
            retourner i
        finsi
    fait
    retourner -1
}
```

Pour l'algorithme récursif, on retourne -1 si l'index dépasse la taille du tableau et la clé n'est pas trouvée :

```
fonction parcours_recursif(E: arbre [n] entier, cle, k = 1 entier, S: entier) {
    si arbre[k] == cle alors
        retourner k
    sinon si k > n alors
        retourner -1
    finsi
    retourner parcours_recursif(arbre, cle, k + 1)
}
```

II. Parcours dichotomique (ordonné) dans l’arbre binaire de recherche

1) Algorithme binaire

```
fonction parcours_dichotomique(E: arbre [n] entier, cle entier, S: entier) {
    k = 1
    tant que k <= n faire
        si cle == arbre[k] alors
            retourner k
        sinon si cle < arbre[k] alors
            k = k * 2
        sinon
            k = k * 2 + 1
        fsi
    fait
fin
```

2) Pire des cas et complexité

Le pire des cas correspond à la situation où la clé recherchée est dans la deuxième partie du tableau. Dans ce cas, la complexité en O de l'algorithme est (O(\log_2 n)).

3) Meilleur des cas et complexité

Le meilleur des cas correspond à la situation où la clé recherchée est la tête de l'arbre binaire (premier élément du tableau). Dans ce cas, la complexité en O de l'algorithme est (O(1)).

4) Algorithme récursif

Voici le même algorithme sous forme récursive :

```
fonction parcours_dichotomique_recursif(E: arbre [n] entier, cle, k = 1 entier, S: ent
    si cle == arbre[k] alors
        retourner k
    finsi
    si cle < arbre[k] alors
        retourner parcours_dichotomique_recursif(arbre, cle, k * 2)
    finsi
    retourner parcours_dichotomique_recursif(arbre, cle, k * 2 + 1)
}
```

5) Modification des versions si la clé n'appartient pas au tableau

Pour l'algorithme itératif, on retourne simplement -1 à la fin de la boucle si la clé n'est pas trouvée :

```
tant que k <= n faire
    si cle == arbre[k] alors
        retourner k
    sinon si cle < arbre[k] alors
        k = k * 2
    sinon
        k = k * 2 + 1
    fsi
fait
retourner -1
```

Pour l'algorithme récursif, on retourne -1 si l'index k dépassent la taille et la clé n'est pas trouvée :

```
fonction parcours_dichotomique_recursif(E: arbre [n] entier, cle, k = 1 entier, S: ent
    si k > n alors
        retourner -1
    finsi
    si cle == arbre[k] alors
        retourner k
    finsi
    si cle < arbre[k] alors
        retourner parcours_dichotomique_recursif(arbre, cle, k * 2)
    finsi
```

```
        retourner parcours_dichotomique_recursif(arbre, cle, k * 2 + 1)
    }
```