

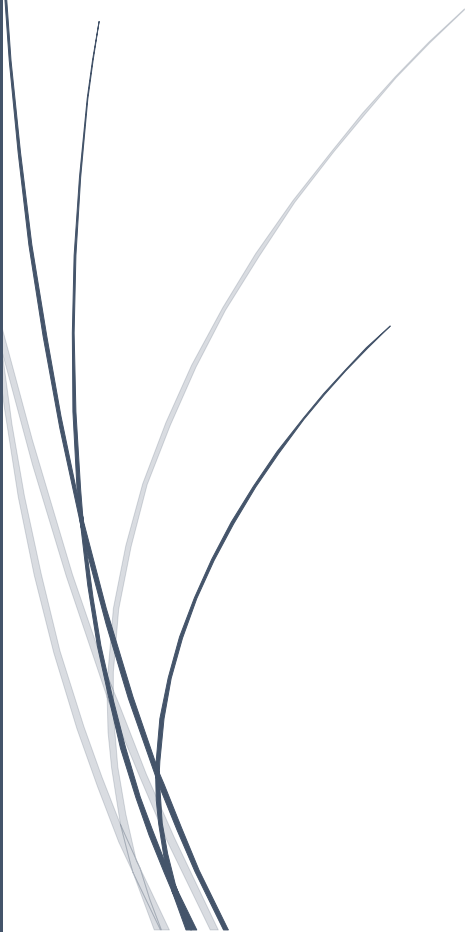
A dark blue vertical bar runs along the left edge of the page. A blue arrow points to the right from this bar, containing the date.

17/09/2022

# Travail #3

## Algorithme et efficacité

Document d'analyse

Several thin, curved lines in dark blue and light grey originate from the bottom left and sweep upwards and to the right.

Camille Boucher - BOUC19590002  
UQAC

## Table des matières

Projet #1 Efficacité .....	1
Explications des résultats des $O(n)$ du tableau précédent : .....	1
Analyse et conclusion : .....	3
Projet #2 Algorithmique .....	4
Mise en situation 1 .....	4
Mise en situation 2 .....	5
Mise en situation 3 .....	7

### Projet #1 Efficacité

Le but de ce projet est d'étudier et de comprendre la notion d'efficacité des algorithmes. Pour cela nous avons implémenté 6 fonctions. Ces fonctions comportent notamment des boucles avec une variable « n ». Nous avons ci-dessous un tableau récapitulatif des temps d'exécution des fonctions en millisecondes en fonction. Il y a également dans la dernière colonne l'ordre d'exécution  $T_f(n)$  dont les calculs sont détaillés plus bas.

Fonction	n=10	n=100	n=1000	n=10000	$O(n)$
$f_1$	0.0002	0.0003	0.0013	0.0125	$n$
$f_2$	0.0003	0.0133	1.2082	121.012	$n^2$
$f_3$	0.0020	2.1229	2149.933	2132888	$n^3$
$f_4$	0.0004	0.0099	0.6067	62.9017	$n^2$
$f_5$	0.0113	1264.49	Non réalisable	Non réalisable	$n^5$
$f_6$	0.0031	15.8143	150202.9	Non réalisable	$n^4$

*Table des temps d'exécution de fonction de forme  $f(n)$  en fonction de n et mesuré en millisecondes*

Explications des résultats des  $O(n)$  du tableau précédent :

```
void f1(int n) {
    int somme = 0;           //c0
    for (int i = 0; i < n; i++) //n
        somme++;             //c1
}
```

$$T_{f_1}(n) = c^1 n + c^0 \text{ donc } O(f_1) = n.$$

```
void f2(int n) {
    int somme = 0;           //c0
    for (int i = 0; i < n; i++) //n
        for (int j = 0; j < n; j++) //n
            somme++;           //c1
}
```

$$T_{f_2}(n) = c^1 n^2 + c^0 \text{ donc } O(f_2) = n^2.$$

```
void f3(int n) {
    int somme = 0;           //c0
    for (int i = 0; i < n; i++) //n
        for (int j = 0; j < n * n; j++) //n^2
            somme++;           //c1
}
```

$$T_{f_3}(n) = c^1 n^3 + c^0 \text{ donc } O(f_3) = n^3.$$

```

void f4(int n) {
    int somme = 0;           //c0
    for (int i = 0; i < n; i++) //n
        for (int j = 0; j < i; j++) //n
            somme++;          //c1
}

```

Nous avons deux boucles « for », avec la première on sait que  $i = O(n)$  donc  $T_4(n) = c^1 n^2 + c^0$ .

On peut vérifier avec des exemples de valeur et une expression mathématiques :

i	j	Action élémentaire
0	\	\
1	0	1
2	0,1	2
3	0,1,2	3

En fait,  $T_{f_4}(n) = c^0 + c^1 \sum_{i=0}^{n-1} i = c^0 + c^1 \frac{n(n-1)}{2}$  donc on a bien  $O(f_4) = n^2$ .

```

void f5(int n) {
    int somme = 0;           //c0
    for (int i = 0; i < n; i++) //n
        for (int j = 0; j < i * i; j++) //n^2
            for (int k = 0; k < j; k++) //n^2 car j = O(n^2)
                somme++;          //c1
}

```

Nous avons trois boucles « for », avec la première on sait que  $i = O(n)$  donc j et k sont des  $O(n^2)$ .

On peut vérifier avec des exemples de valeur et une expression mathématiques :

i	j	k	Action élémentaire	Relation observée
0,1	\	\	\	
2	0	\	0	Cout = 6 = somme de j =0 à i <sup>2</sup> -1
	1	0	+1	
	2	0,1	+2	
	3	0,1,2	+3	
3	1	0	+1	Cout = 6 = somme de j =0 à i <sup>2</sup> -1
	2	0,1	+2	
	3	0-2	+3	
	4	0-3	+4	
	5	0-4	+5	
	6	0-5	+6	
	7	0-6	+7	
	8	0-7	+8	

En fait,  $T_{f_5}(n) = c^0 + c^1 \sum_{i=2}^{n-1} \sum_{j=0}^{i^2-1} j = c^0 + c^1 \frac{1}{20} n(n+1)(n-1)(n-2)(2n-1)$  donc on a bien  $O(f_5) = n^5$ .

```

void f6(int n) {
    int somme = 0; //c0
    for (int i = 1; i < n; i++) //n
        for (int j = 1; j < i * i; j++) //n^2
            if (j % i == 0) //c1
                for (int k = 0; k < j; k++) //n, nbJ = nbDiviseursDei <= racine(i)/2 < racine(O(n^2))/2 = O(n)
                    somme++; //c2
}

```

Nous avons trois boucles « for », avec la première on sait que  $i = O(n)$  donc  $j = O(n^2)$ . En revanche  $k = O(n)$ , car k correspond au nombre de diviseur de j. Or lorsqu'on cherche un à un les diviseurs d'un nombre, on compte le nombre de diviseurs jusqu'à la racine carrée du dividende et on multiplie le nombre trouvé par 2. Soit  $k \leq \frac{\sqrt{j}}{2} = \frac{\sqrt{O(n^2)}}{2} = O(n)$  par règle de simplification.

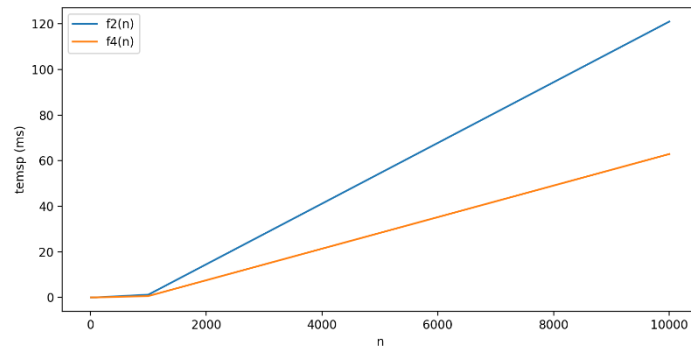
On peut vérifier avec des exemples de valeur et une expression mathématiques :

i	j	k	Action élémentaire	Relation observée
0,1	\	\	\	
2	0	\	2	i
	1	\		
	2	0,1		
	3	\		
3	1	\	3 +6	i +2i
	2	\		
	3	0-2		
	4	\		
	5	\		
	6	0-5		
	7	\		
	8	\		

En fait,  $T_{f_5}(n) = c^0 + c^1 \sum_{i=2}^{n-1} \sum_{j=0}^{i-1} ij = c^0 + c^1 \frac{1}{24} n(n-1)(n-2)(3n-1)$  donc on a bien  $O(f_6) = n^4$ .

Analyse et conclusion :

On peut voir que globalement les **résultats d'implémentation correspondent à ceux prédis par l'analyse** : pour la fonction  $f_3$  c'est assez flagrant le résultat est multiplié par 1000 à chaque puissance de 10 de n (cohérent car  $O(f_3) = n^3$ ). On peut toutefois que les résultats pour n=10 dénotent un peu de la tendance linéaire, on peut regarder ça sur  $f_2$  et  $f_4$ , qui sont des  $O(n^2)$ , sur la courbe ci-dessous : à partir de n=1000 on a une belle ligne droite, mais les résultats à n=10 provoquent une rupture pentes. Ces résultats pour n petit sont un peu plus lents que prévu, mais c'est vraiment à n grand que l'on peut voir la tendance d'évolution.



*Tracé des temps d'exécution de f2 et f4*

Pour f5 et f6 nous n'avons pas pu calculer certains temps d'exécution, qui auraient été selon l'analyse de plus d'un jour ou même d'une semaine ; c'est dans ce cas qu'on peut voir l'importance de l'efficacité et l'optimisation des fonctions traitant beaucoup de données, comme des bases de données ou des données lourdes (photos, vidéos haute définition).

## Projet #2 Algorithmique

### Mise en situation 1

```

/*
Principe de la méthode de recherche :
- selon la partie de l'élément recherché, tronquer une partie du tableau initial
(garder que les pairs ou impairs)
- appliquer l'algorithme de dichotomie sur le tableau tronqué (qui est trié)
*/

//méthode de recherche dichotomique
//retourne la position de k dans un tableau "trie" (cf énoncé) de taille n
int dichotomie(int tab[], int n, int k)
{
    int l = 0; //gauche
    int r = n-1; //droite
    while (l <= r)
    {
        int m = (l + r) / 2;
        if (tab[m] < k) { l = m + 1; }
        else if (tab[m] > k) { r = m - 1; }
        else { return m; }
    }
    return -1; //k is not in tab
}

//pseudo code pour la mise en situation
//retourne la position de k dans un tableau "trie" (cf énoncé) de taille n (n est
forcément impair)
int recherche1(int tab[], int n, int k)
{
    vector<int> sousTab; //c0
    int m = 0; //c1//taille de la sous table que nous utiliserons pour la recherche
dichotomique
    if (k % 2 == 0) //c2//si k est pair, on va tronquer la partie impaire et ne
garder que les éléments pairs

```

```

{
    sousTab = [tab[0] à tab[n + 1 / 2]]; //c3// tab[n+1/2] correspond au max,
    //on s'en moque s'il est pair ou impair, chercher sa parité pour
    l'inclure ou l'exclure rajouterait de la complexité
    m = (n + 1) / 2 + 1; //c4//taille de sousTab
    return dichotomie(sousTab, m, k); //sachant que sousTab est trié on peut
    appliquer la dichotomie
}

else // k impair //c2
{
    sousTab = [tab[n + 1 / 2] à tab[n-1]]; //c3//tab[n+1/2] correspond au max,
    on inverse sousTab ; //c5// sousTab était trié dans l'ordre décroissant,
    on le veut dans l'ordre croissant
    m = taille de sousTab; //c4
    return dichotomie(sousTab, m, k); // //sachant que sousTab est trié on peut
    appliquer la dichotomie
}
}

```

Analyse :

#Pire cas

On sait tout d'abord que la recherche dichotomique dans un tableau de taille  $n$  est un  $O(\log(n))$  dans le pire cas.

Donc pour le pire cas, correspondant au cas où l'élément recherché est impair (car on doit inverser le tableau) et n'est pas dans le tableau, on a  $T(f(n)) = c_0 + c_1 + c_2 * (c_3 + c_4 + c_5 + \log((n-1)/2))$ . Donc dans le pire cas  $T(f(n)) = O(\log(n))$ .

#Meilleur cas

Dans la recherche dichotomique, le meilleur cas correspond au cas où l'élément recherché est pair et correspond à l'élément situé à la moitié du tableau. Dans ce cas,  $T(f(n)) = c_0 + c_1 + c_2 * c_3 * c_4 * \log(c)$ . Donc dans le meilleur cas  $T(f(n)) = O(1)$ .

#Cas moyen

Dans la recherche dichotomique, le cas moyen correspond à un cas où l'élément est quelque part dans le tableau, vers le quart ou trois-quarts. Dans ce cas, on est à une constante du pire cas donc  $O(\log(n))$ .

## Mise en situation 2

/\*

Principe de la méthode de d'insertion :

- on utilise l'algorithme précédent pour chercher la position à laquelle il faut insérer l'élément pour respecter le tri du tableau, toutefois on va modifier la fonction dichotomie en « plusProcheDichotomie », pour qu'elle nous renvoie non pas la position de l'élément dans le tableau mais la position de l'élément après ou avant lequel il faut l'insérer.

- pour respecter le tri et le fait que l'élément max se trouve au milieu, il faut aussi insérer quelque chose en symétrie  
\*/

//methode de recherche "dichotomique"

//retourne la position la plus proche de laquelle on doit insérer k dans un tableau "trie" (cf enonce) de taille n en utilisant le principe de recherche dichotomique

int plusProcheDicho(int tab[], int n, int k)

```
{
    int g = 0; //gauche
    int d = 0; //droite
    int plusProcheIndex = 0;
    while (g <= d)
    {
        plusProcheIndex = (g + d) / 2;
        if (tab[plusProcheIndex] < k) { g = plusProcheIndex + 1; }
        else if (tab[plusProcheIndex] > k) { d = plusProcheIndex - 1; }
        else { return plusProcheIndex; } //il y a deja k dans le tableau on peut
        linsérer avant ou apres
    }
    return plusProcheIndex; //k is not in tab
}
```

//pseudo code pour la mise en situation

void insertion(vector<int> tab[], int n, int k)

```
{
    vector<int> sousTab;
    vector<int> copieTab;
    int m = 0; //c1
    int plusProcheindex = 0;
    if (k % 2 == 0) //si k est pair
    {
        On verifie d'abord si k est plus petit que le premier element, si oui il
        suffit juste de la rajouter au début et recopier le reste du tableau

        Sinon :
        sousTab = [tab[0] à tab[n + 1 / 2]];
        m = (n + 1) / 2 + 1;
        plusProcheindex = plusProcheDicho(sousTab, m, k);
        on declare et instancie un iterateur constant pour le vecteur tab // it =
        v.begin() + plusProcheindex
        tab.insert( iterateur, k)
        puis on ajoute a la fin le dernier element pour avoir une symetrie
    }

    else // k impair
    {
        On verifie d'abord si k est plus petit que le dernier element, si oui il
        suffit juste de la rajouter a la fin du tableau

        Sinon :
        sousTab = [tab[n + 1 / 2] à tab[n-1]];
        on inverse sousTab;
        m = taille de sousTab;
        plusProcheindex = plusProcheDicho(sousTab, m, k);
        on insere à l'aide d'un iterateur et de la methode insert le premier
        element (pour qu'il y soit deux fois pour la symétrie)
        de la même manière on insère k
    }
}
```

Analyse :

#Pire cas

On sait tout d'abord que la méthode « plusProcheDicho » inspirée de la dichotomie dans un tableau de taille  $n$  est un  $O(\log(n))$  dans le pire cas.

Donc le pire cas correspond au cas où l'élément à insérer est impair (car on doit revert le tableau) et n'est pas dans le tableau. Pour l'insertion, le pire cas est que l'on doit faire une réallocation du vecteur `tab` ; donc on a  $T(f(n)) = c_0 + c_1 + \text{constant} + \log((n-1)/2) + n$ . Donc dans le pire cas  $T(f(n)) = O(n)$ .

#Meilleur cas

Le meilleur cas correspond au cas quand l'élément à insérer est impair et plus petit que le dernier élément (pas de recopie) ainsi que le cas où la capacité du vecteur `tab` soit plus grande que sa taille initiale (pour ne pas avoir de réallocation). Dans ce cas, le meilleur cas correspond à la combinaison de ces deux cas plus et  $T(f(n)) = O(1)$ .

#Cas moyen

Dans la recherche dichotomique, le cas moyen correspond à un cas où l'élément est quelque part dans le tableau, vers le quart ou trois-quarts. Et après on doit quand même recopier le tableau. Dans ce cas, on est à une constante du pire cas donc  $T(f(n)) = O(n)$ .

### Mise en situation 3

J'ai codé pour la mise en situation 3 une solution itérative (qui fonctionne) qui donne bien le résultat mais qui ne donne pas le même affichage que l'énoncé.

Je me suis basée sur le fait que :

Nombre	Nombre de fois affiché
2	$8 = 2^3 = 2^{5-2}$
3	$4 = 2^2 = 2^{5-3}$
4	$2 = 2^1 = 2^{5-4}$
5	$1 = 2^0 = 2^{5-5}$

```
int doublefacto(int n)
{
    int resultat = n;
    cout << "\n" << resultat;
    for (int i = n-1; i > 1; i--) {
        for (int j = 1; j <= pow(2, n-i); j++) {
            cout << "\n" << i;
            resultat = resultat * i;
        }
    }
    return resultat;
}
```



Ici, le pire cas, le cas moyen et le meilleur cas sont les mêmes (toujours le même « chemin » dans l'algorithme). On a  $T(f(n)) = O(2^n n)$ .