TP2 Vasileios Skarleas et Yanis Sadoun

Exercice 1

Q1-1 Algorithme et complexité

On initialise `min` et `max` en leur assignant la valeur du premier élément du tableau. Ensuite, on parcourt le reste du tableau, ce qui représente n-1 éléments, en comparant chaque élément aux valeurs de base de `min` et `max`. Si un élément est inférieur à `min` ou supérieur à `max`, on met à jour `min` ou `max` en conséquence.

La complexité de l'algorithme est O(2(n-1)).Plus precisement:

```
void get_min_max_1(int *tab, int n, int *min, int *max)
{
    int max_local = tab[0]; //O(1)
    int min_local = tab[0]; //O(1)

    for (int i = 1; i < n; i++) // (n-1) x  3*O(1) [car il fait deux
comparaisons et en pire des cas il va metre à jour le min_local ou le
max_local]
    {
        if (tab[i] < min_local) //O(1)
        {
            min_local = tab[i]; //O(1)
        }
        else if (tab[i] > max_local) //O(1)
        {
            max_local = tab[i]; //O(1)
        }
    }
    *min = min_local; //O(1)
    *max = max_local; //O(1)
}
```

Ainsi, la complexité totale est O(3 * (n-1)) ou encore O(2 * (n-1)) qui ce simplifie en O(n). La difference entre O(2 * (n-1)) et O(3 * (n-1)) est une constante qui est negligable. On peut avoir exactement une comlexité en O(2 * (n-1)) dans le cas ou :

```
if (tab[i] < min_local) //O(1)
{
min_local = tab[i]; //O(1)
}
else
{
max_local = tab[i]; //O(1)
}
```

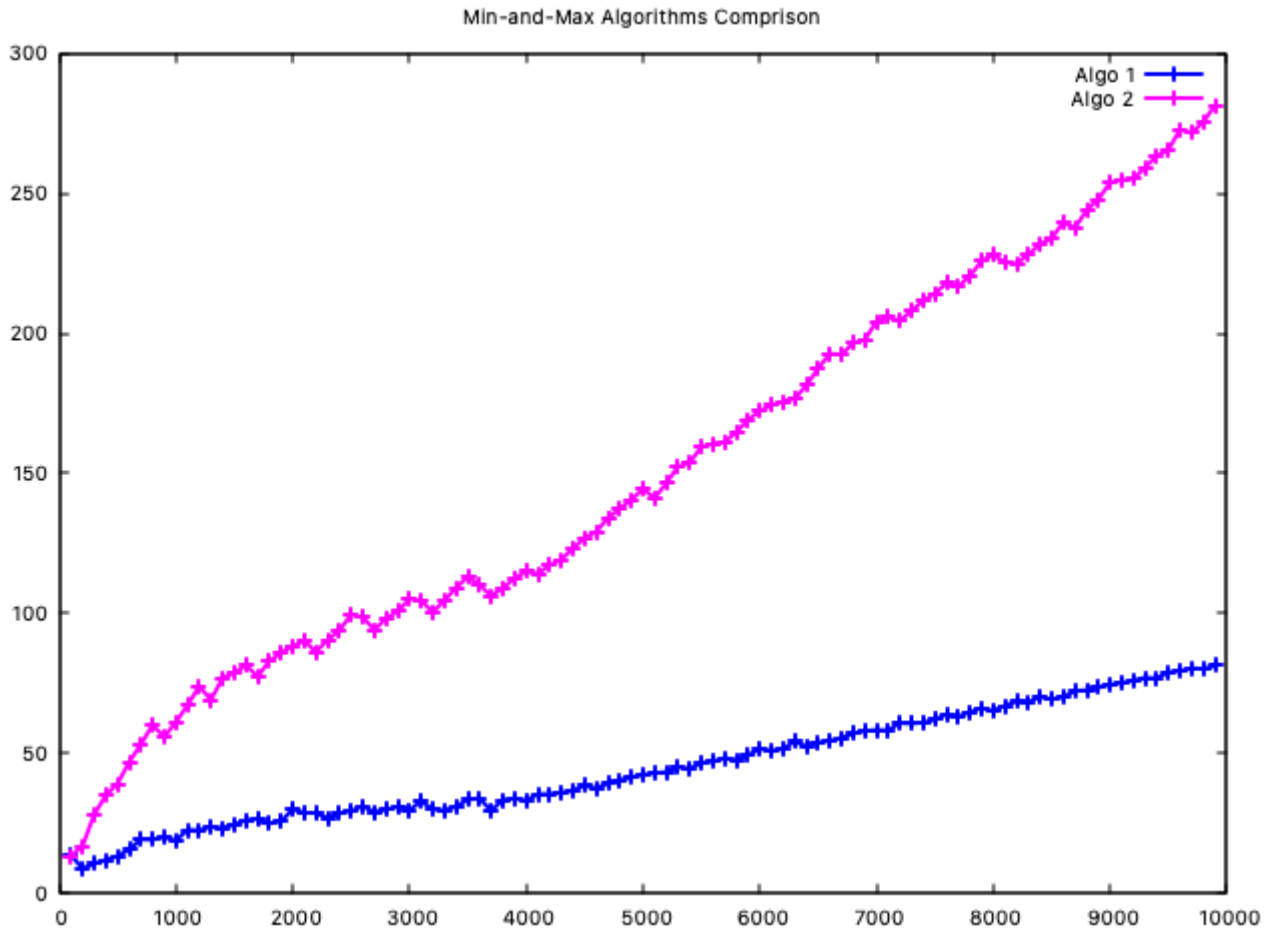
Q1-2 Algorithme et complexité

Chaque appel récursif divise le problème en deux sous-problèmes, ce qui implique une complexité de O(log(n)) pour le processus de division. En ce qui concerne la partie de comparaison, nous avons soit un cas de base avec un seul élément, soit une comparaison entre deux éléments suivie d'un retour à l'étape de division précédente.

Par conséquent, nous avons une compléxité en O(3 * log(n)) pour l'ensemble des appels récursifs et O(n - 2) pour les comparaisons entre deux éléments, à condition que n soit une puissance de 2. En somme, la complexité totale est de O(3 * log(n)) + O((n - 2)), qui peut être simplifiée en O((3/2)n - 2) car log(n) < n.

```
void get_min_max_rec(int *tab, int from, int to, int *min, int *max)
{
    if (from == to)
    {
        // Base case: single element
        *min = *max = tab[from];
    }
    else if (from < to - 1)
    {
        int mid = (from + to) / 2;
        get_min_max_rec(tab, from, mid, min, max);
        get_min_max_rec(tab, mid + 1, to, min, max);
    }
    else
    { // Handle two elements
        if (tab[from] < tab[to])
        {
            *min = tab[from];
            *max = tab[to];
        }
        else
        {
            *min = tab[to];
            *max = tab[from];
        }
    }
}

void get_min_max_2(int *tab, int n, int *min, int *max)
{
    *min = INT_MIN;
    *max = INT_MAX;
    get_min_max_rec(tab, 0, n - 1, min, max);
}
```

Q1-3 Comparaison et conclusions



Il est observé que l'algorithme `get_min_max_1` est plus performant que `get_min_max_2`, ce qui peut sembler contre-intuitif au regard des complexités théoriques, car 2n-1 est supérieur à (3/2)n - 2. L'explication pourrait résider dans les détails d'implémentation et le coût des opérations individuelles.

En effet,

`get_min_max_1` parcourt le tableau une fois, effectuant des comparaisons et des affectations à temps constant. Sa complexité est O(n), ce qui le rend efficace pour des ensembles de données plus petits grâce à son approche simple et directe.

D'autre part , `get_min_max_2` divise récursivement le problème en sous-problèmes plus petits, conduisant à une complexité logarithmique (O(log n)). Cependant, cela implique également des appels de fonctions, un changement de contexte et des comparaisons supplémentaires dans le cas de base, ce qui peut ajouter une surcharge pour les ensembles de données plus petits.

Voici donc , le recapitulatif de nos resultats:

Algorithme	Complexité	Avantages	Inconvenients
`get_min_max_1`	O(n)	Simple, efficace pour les petits ensembles de données	Peut être lent pour les grands ensembles de données
`get_min_max_2`	O(log n)	Asymptotiquement plus rapide pour les grands ensembles de données	Plus de surcharge pour les petits ensembles de données

Pour les très grands ensembles de données (> 10 000 éléments), l'approche diviser pour régner de `get_min_max_2` pourrait surpasser `get_min_max_1` en raison de sa complexité asymptotique moindre. Cependant, `get_min_max_2` peut utiliser un peu plus de mémoire en raison de la surcharge de la pile de récursion.

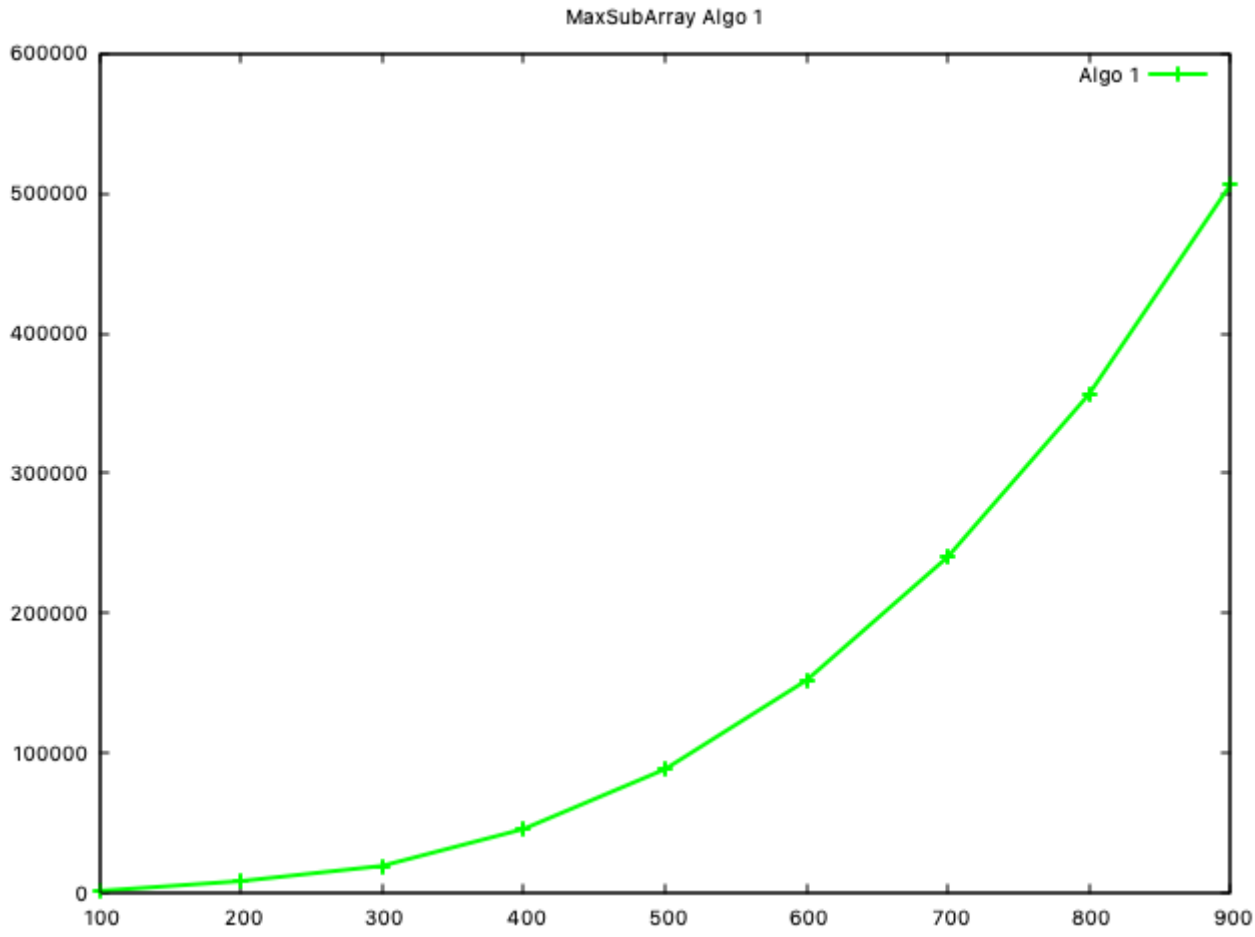
Exercice 2

Q2-1 Algorithme, complexité et temps

Le codage de l'agorithme de sous-tableau de poids maximum est le suivant:

```
int maxSubArraySum1(int *tab, int n)
{
   int max;
   get_max(tab, n, &max);
   int temp = 0;
   for (int i = 0; i < n; i++)
   {
      for (int j = i; j < n; j++)
      {
         for (int k = i; k <= j; k++)
         {
            temp = temp + tab[k];
         }
         if (temp > max)
         {
            max = temp;
         }
      }
   }

   return max;
}
```

L'algorithme considéré a une complexité de O(n^3) car, dans le pire des cas, il comporte trois boucles imbriquées, chacune traitant n éléments. Par conséquent, le nombre total d'opérations est proportionnel à n * n * n = n^3. La courbe représentant le temps d'exécution en fonction de la taille du tableau est bien une courbe cubique. Toutefois On limite ici les tests à un nombre minimum d'éléments, car les temps d'exécution deviennent très importants même pour des tailles relativement petites.



Q2-2 variante en Θ(n^2)

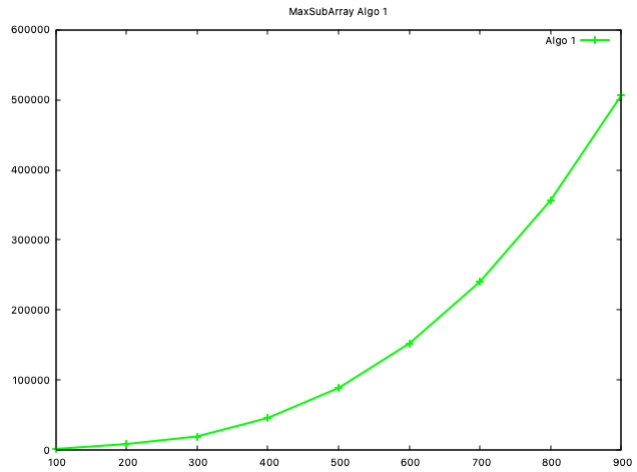
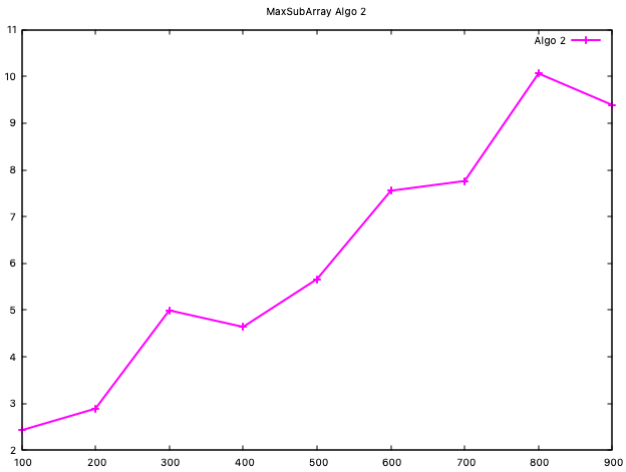
Selon la théorie mathématique, la meilleure solution pour trouver la sous-séquence maximale est celle proposée par l'algorithme de Kadane. L'algorithme proposé par le mathématicien est (notre variante):

```
def max_subarray(numbers):
    """Find the largest sum of any contiguous subarray."""
    best_sum = - infinity
    current_sum = 0
    for x in numbers:
        current_sum = max(x, current_sum + x)
        best_sum = max(best_sum, current_sum)
    return best_sum
```

En effet, si l'élément actuel en question est supérieur à la somme actuelle du sous-tableau (max_current sur le code), alors il est préférable de démarrer un "nouveau" sous-tableau avec tab[i] comme seul élément (il est déja le plus grand dans ce cas là).

À première vue, l'algorithme semble avoir une complexité de O(n). Cependant, il existe une probabilité, bien que faible, qu'il ne soit pas linéaire pour toutes les configurations possibles de données. Néanmoins, on sait que la limite supérieure (le majorant) de sa complexité, dans le pire des cas, est de Θ(n^2).

MaxSumArray1 vs MaxSumArray2	**MaxSumArray2**

MaxSumArray1 vs MaxSumArray2	MaxSumArray2
	

On constate tout de même que les valeurs de temps ne sont pas très précises.

Q2-3 Calculant stm3 en Θ(n)

Pour ce problème on veut faire une divison du tableau, alors mathématiquement stm3 est egal:

$$\max \sum_{\kappa=i}^{m} A[\kappa] + max \sum_{\kappa=m+1}^{j} A[\kappa]$$

$$avec\ j \geq m+1$$

Considérons un tableau A que nous divisons en deux parties, A1 à gauche et A2 à droite, séparées au milieu. stm1 représente la somme maximale du sous-tableau dans la partie gauche A1, et stm2 est celle de la partie droite A2. stm3 est calculé pour inclure les éléments qui chevauchent le milieu, englobant les derniers éléments de A1 et les premiers éléments de A2. Ensuite, nous résolvons le problème récursivement pour stm1 et stm2, ce qui nous donne max1 et max2 respectivement. La dernière étape est la fusion des résultats : nous cherchons le maximum entre max1, max2 et stm3. Le résultat final sera la somme maximale du sous-tableau pour le tableau combiné A.

Complexité

On calcule $\sum_{\kappa=i}^{m} A[\kappa]$ avec une boucle en Θ(n). De plus, on calcule $\sum_{\kappa=m+1}^{j} A[\kappa]$ avec une boucle en Θ(n). On somme les deux resultats. stm(A) = max{max1, max2, stm3}. La complexité de fusion est donc lie au calcul de stm3 qui ce fait en Θ(n).

Q2-4

D'après la question précedente, on propose l'agorithme suivante:

1.On divise le tableau en deux sous-tableaux, gauche (left) et droite (right). Il s'agit d'une appel recursive ayant comme cas de base: un tableau d'un seul élément qui renvoie cet élément comme la somme maximale de ce sous-tableau.

2.On calcule stm3, qui représente la somme maximale du sous-tableau chevauchant le milieu, en trouvant la somme maximale dans le sous-tableau de gauche (tout en sauvegardant le resultat au dernier element du tableau de gauche) et la somme maximale dans le sous-tableau de droite (tout en sauvegardant le resultat au premier element du tableau de droite), puis en les additionnant = c'est le stm3.

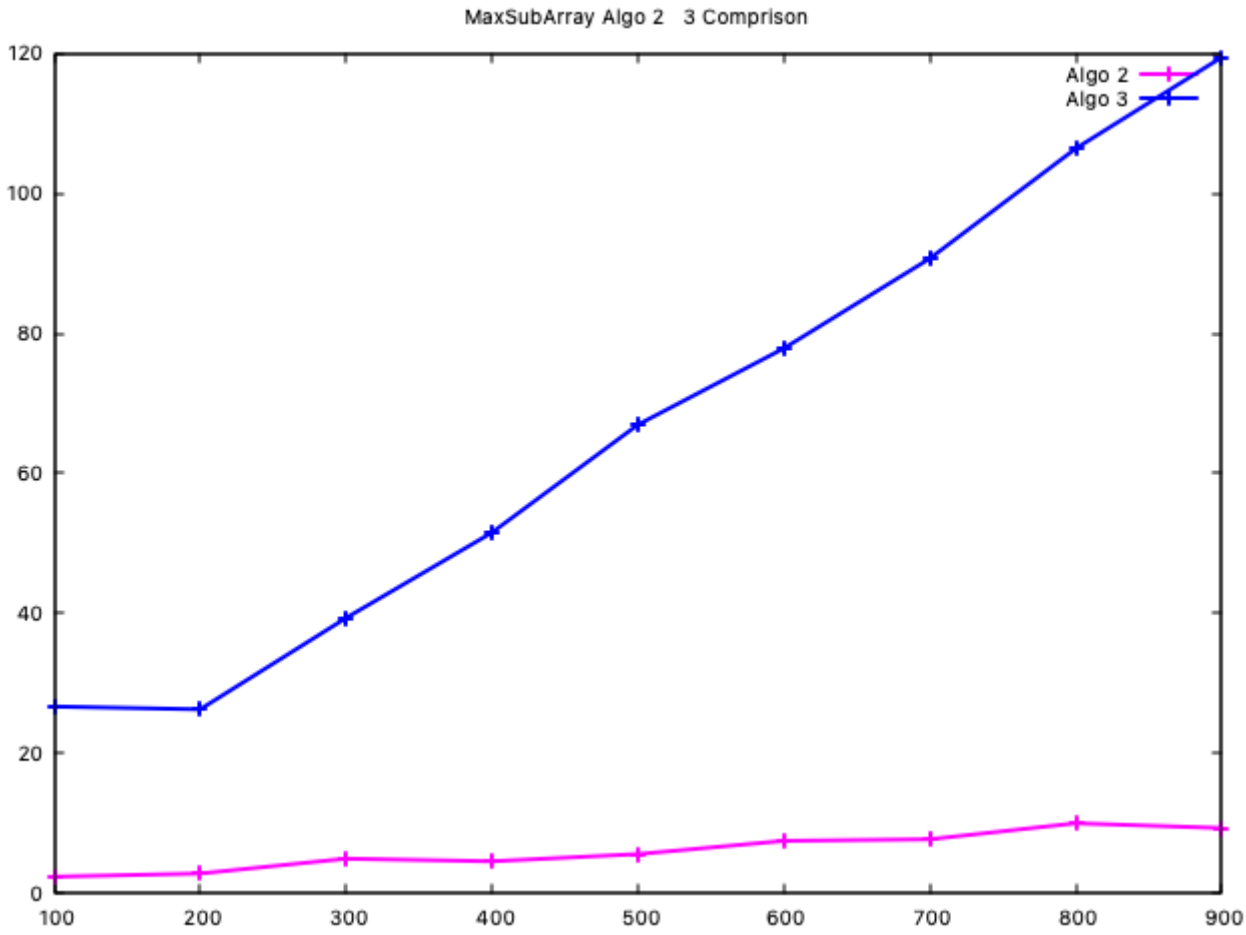
3.On effectue une comparaison pour déterminer le maximum entre stm1, stm2, et stm3.

Complexité

En combinant ces deux parties, la complexité totale de l'algorithme est déterminée par le produit du nombre de niveaux de récursion et du coût de chaque opération de fusion. Puisque l'algorithme réalise une opération de fusion linéaire O(n) à chaque niveau de récursion et qu'il y a O(log n) niveaux, la complexité totale de l'algorithme est O(n log n).

De même, d'après le theoreme-maitre on a :

`T(n) = 2T(n/2) + O(n)` avec a = 2, b = 2 et d = 1 => compléxité en O(n log(n))



En comparant les deux algorithmes, on observe que algo2 est plus efficace que algo3 pour un nombre donné d'itérations. En effet, algo2 présente une complexité de O(n) tandis que algo3 a une complexité de O(n log n) .En effet,pour tout n positif, n est toujours inférieur à nlogn. Cela implique que, pour des tailles de tableau identiques, algo2 sera plus rapide que algo3 en raison de sa complexité temporelle inférieure.

Q2-5

Ca serait faux, car on peut avoir:

pref(A1) =/= pref(A) et suff(A2) =/= suff(A).

Par exemple,pour un tableau `1 −2 3 −1` avec A1 = {1,-2} et A2 = {3, -1} , on a pref(A1) = 1 et pref(A2) = 3 et pref(A) = 3. Donc on est biens dans le cas de pref(A1) =/= pref(A).

Q2-6

Formule

La formule pour déterminer le quadruplet est la suivante:

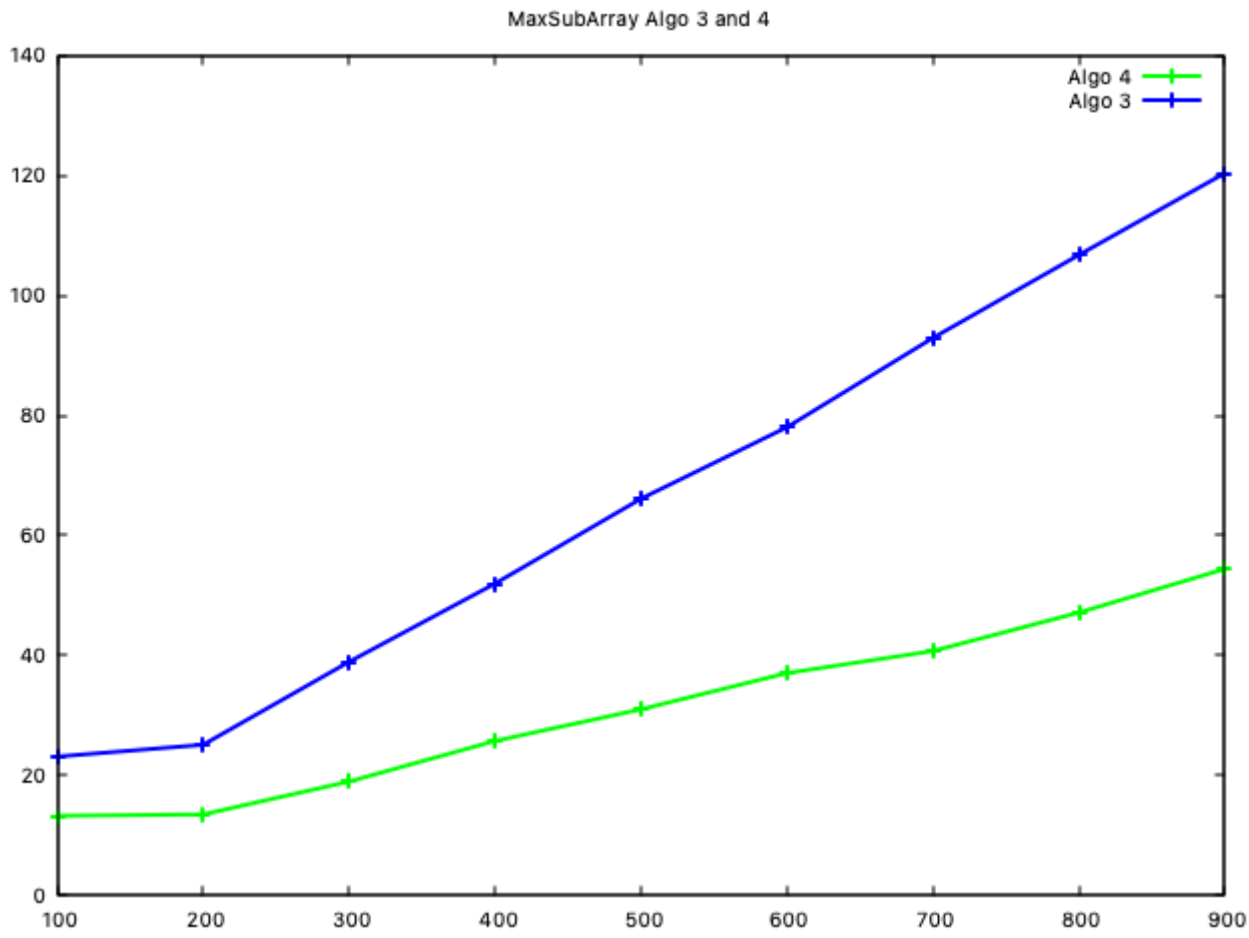
(**max{stm1, stm2, suff1 + pref2}** , *max{pref1, tota1 + pref2}*, max{suff2, suff1 + tota2}, tota1 + tota2). De plus on remarque que la compelxité de cette operation est Θ(1).

Complexité

D'apres le théorème maitre on a:

`T(n) = 2T(n/2) + θ(1)` car tous les operations de comparison qui sont effectués sur maxSubArraySumRec2 sont de complexité O(1). Ainsi a = 2, b = 2 et d = 0;

Ainsi la complexité est Θ(n) .



D'après les résultats obtenus par la comparaison graphique, l'algorithme 4, qui présente une complexité de Θ(n), est plus efficace que l'algorithme 3, dont la complexité est de O(nlogn). Les résultats théoriques sont donc corroborés par les expérimentations.

Exercice 3

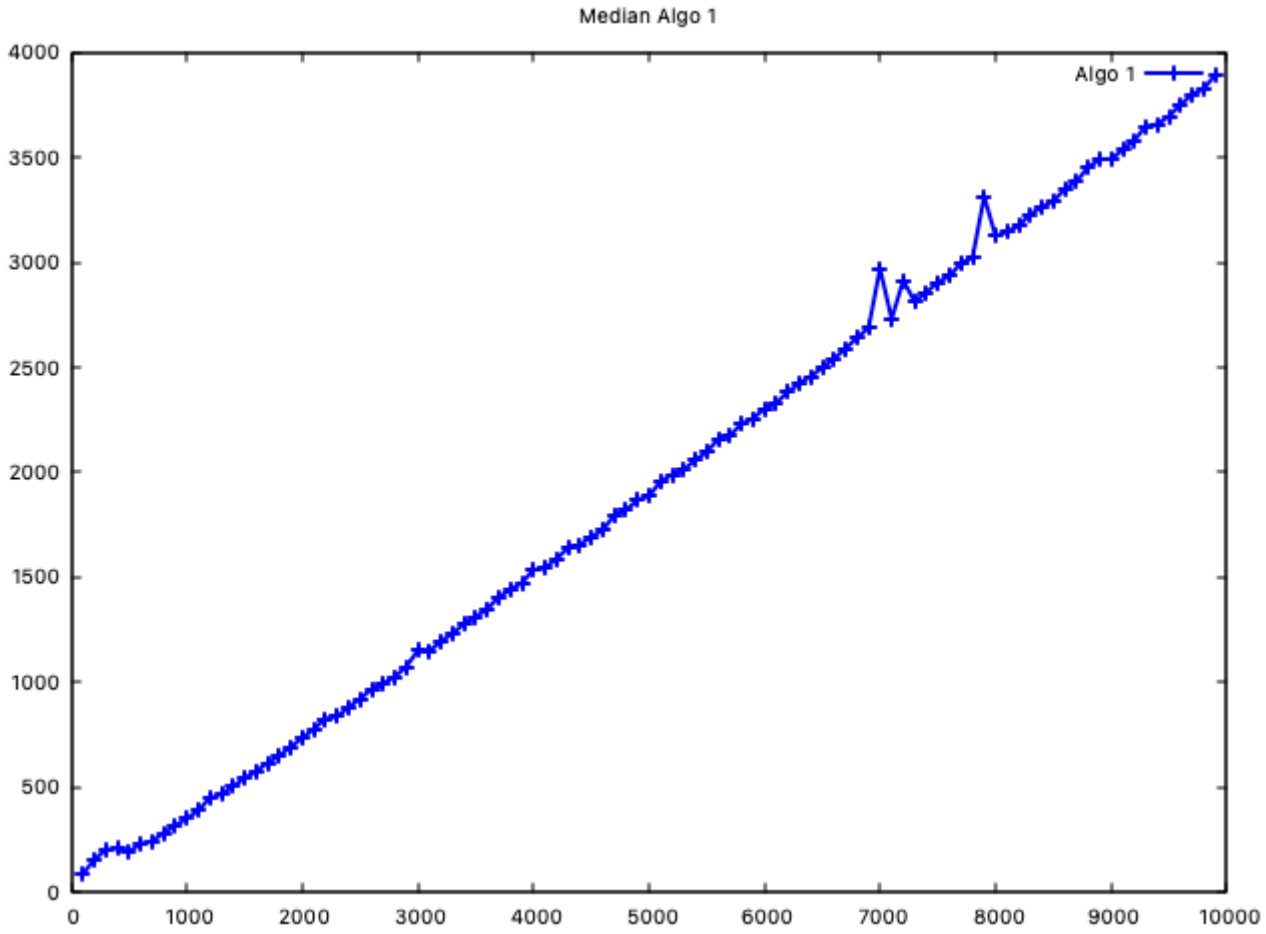
Q3-1 Médiane en utilisant tri fusion

L'agorithme est assez simple:

```
void median(int *tab, int n)
{
    tri_fusion(tab, n); //O(n log(n))
    if (n % 2 == 0) //O(1)
    {
        printf("Median number are: %d and %d\n", tab[(n/2)-1], tab[n/2]);
//O(1)
    }
    else
    {
        printf("Median number is: %d\n", tab[n/2]); //O(1)
```

```
    }
}
```

Dès lors, sa complexité est O(n log(n)) et sa courbe de temps d'exécution est la suivante:



Q3-2

Il faut trouver un algorithme selon les spécifications suivantes:

> Développer une méthode de type diviser pour régner pour ce problème. Soit selection(S,k) la fonction qui sélectionne le keme plus petit élément de S. Pour ce faire, on utilise un élément pivot v du tableau en fonction duquel on partitionne S en trois sous-ensembles : le sous-ensemble SG des éléments plus petits que v, le sous-ensemble Sv des éléments égaux à v et le sous-ensemble SD des éléments plus grands que v. Si |SG| < k ≤ |SG| + |Sv| alors v est le keme plus petit élément de S. Identifier l'appel récursif à réaliser après partition de S en SG, Sv et SD.

Dès lors, on propose l'algorithme suivant:

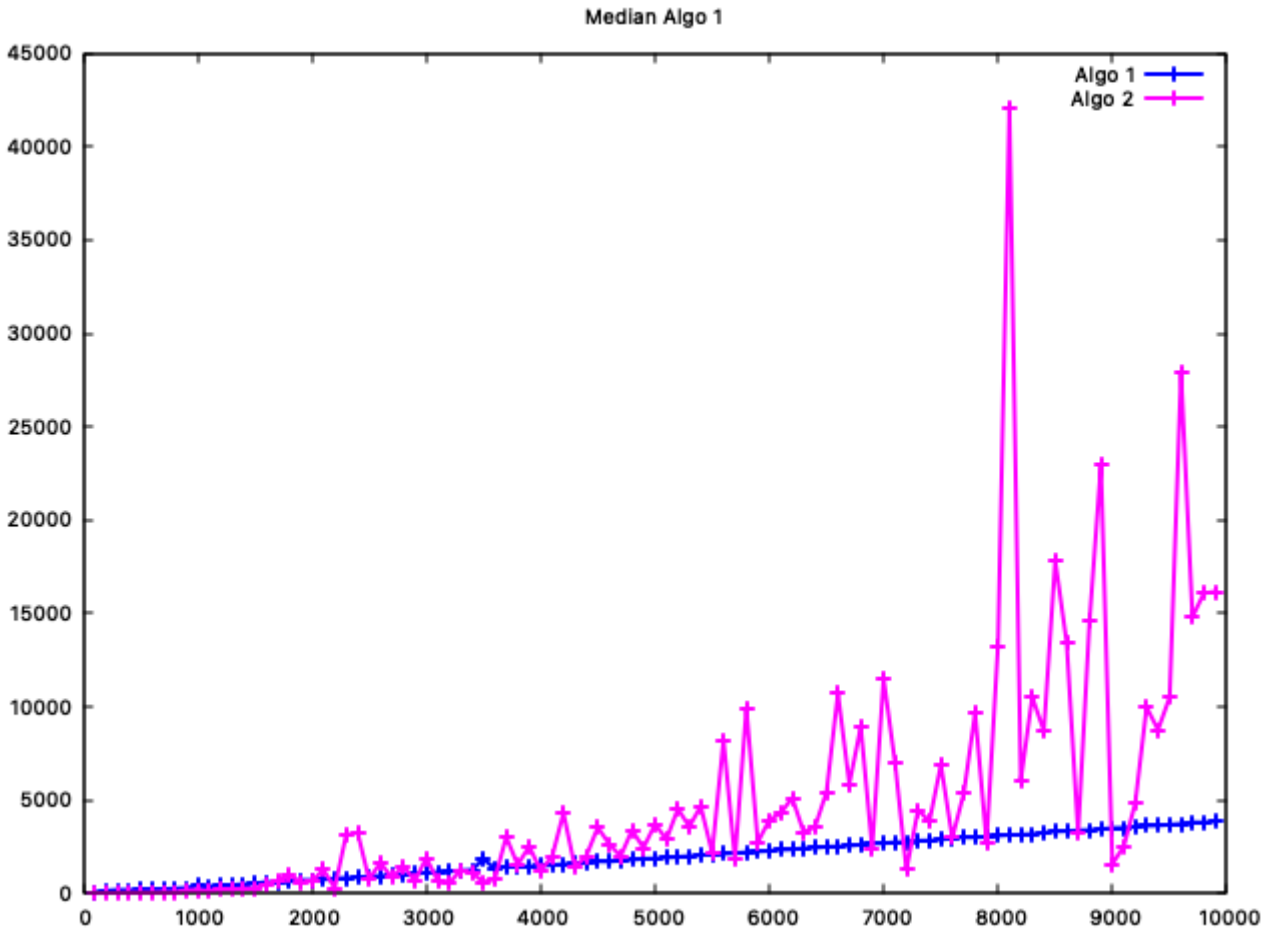
```
fonction median_2 (tableau, n)
    Entrée: un tableau d'entiers et sa longueur
    Sortie: la médiane du tableau

si longeur = 0 alors
    mediane N'EXISTE PAS (NULL)
sinon
```

```
    si n est impair alors
        retourner smallest_k(tableau, 0, n-1, n/2)
    sinon
        retourner (smallest_k(tableau, 0, n-1, n/2 -1) +
smallest_k(tableau, 0, n-1, n/2))/2
    finsi
finsi

## Basé sur l'idée classique que si l'échantillon (trié) est d'un nombre
impair, alors la médiane est l'élément central, et dans le cas d'une
longueur paire, c'est la moyenne des deux nombres centraux.

fonction smallest_k(tableau, debut, fin, k)
    Entrée : un tableau d'entiers, début et fin du traitement du sous-
tableau du tableau original, ainsi qu'un nombre k.
    Sortie : Retourne le k-ième plus petit élément du sous-tableau [debut,
..., fin].

si debut = fin
alors
            retourner tableau[debut] ## Il y a un seul élément dans le
tableau, donc il est effectivement le k-ième élément le plus petit.
    finsi

position_de_pivot <- pivot_division(tableau, debut, fin)

si k = position_pivot - debut alors
    retourner tableau[position_de_pivot] #kieme element le plus petit
trouvé
sinon si k < position_pivot - debut alors
    retourner smallest_k(tableau, debut,position_de_pivot - 1, k) #le
kieme element le plus petit n'est pas encore trouvé mais il va être à
gauche
sinon
    retourner smallest_k(tableau,position_de_pivot+ 1, fin, k -
(position_pivot -debut+1)) #le kieme element va être à droite
finsi
```

La fonction pivot_division choisit un pivot aléatoirement et partitionne le sous-tableau du tableau original conformément aux instructions énoncées.

Cet algorithme est inspiré par le cours CS125 de l'automne 2016, Unité 4, donné par le Professeur Jelani Nelson à l'Université Harvard (source: https://people.seas.harvard.edu/~cs125/fall16/lec4.pdf)

La complexité de cet algorithme varie entre O(n log n) et O(n^2), en fonction de la distribution des données et de l'efficacité du pivot choisi. Cette fourchette de complexité a été vérifiée expérimentalement, comme le montrent les résultats graphiques :



Pour conclure, on observe que l'allure de algo2 semble plus rapide que celle d'algo1.

Exercice 4

Le code pour résoudre le puzzle "Shadows of the Knight" est:

```
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/**
 * Auto-generated code below aims at helping you parse
 * the standard input according to the problem statement.
 **/

int main() {
  // width of the building.
  int W;
  // height of the building.
  int H;
  scanf("%d%d", &W, &H);
  // maximum number of turns before game over.
  int N;
  scanf("%d", &N);
  int X0;
```

```
    int Y0;
    scanf("%d%d", &X0, &Y0);

    int x1 = 0;
    int y1 = 0;
    int x2 = W - 1;
    int y2 = H - 1;

    // game loop
    while (1) {
        // the direction of the bombs from batman's current location (U, UR,
R, DR,
        // D, DL, L or UL)
        char bomb_dir[4];
        scanf("%s", bomb_dir);

        bomb_dir[strcspn(bomb_dir, "\n")] = '\0';

        if (strcspn(bomb_dir, "U") != strlen(bomb_dir))
        {
            y2 = Y0 - 1;
        }
        else if (strcspn(bomb_dir, "D") != strlen(bomb_dir))
        {
            y1 = Y0 + 1;
        }

        if (strcspn(bomb_dir, "L") != strlen(bomb_dir))
        {
            x2 = X0 - 1;
        }
        else if (strcspn(bomb_dir, "R") != strlen(bomb_dir))
        {
            x1 = X0 + 1;
        }

        X0 = x1 + (x2 - x1) / 2;
        Y0 = y1 + (y2 - y1) / 2;

        // the location of the next window Batman should jump to.
        printf("%d %d\n", X0, Y0);
    }

    return 0;
}
```

Nous avons créé deux paires de coordonnées, (x1, y1) et (x2, y2), qui définissent les coins diagonaux d'un sous-tableau 2D sur lequel nous effectuons nos opérations. Lors de la lecture d'une série de commandes, nous cherchons les caractères "U" (Up), "D" (Down), "L" (Left) et "R" (Right) et effectuons les déplacements de coordonnées nécessaires en conséquence. La dernière étape consiste à mettre à jour les coordonnées du caractère contrôlé par les variables (x0, y0) selon ces déplacements.

Phrase	x	y
U		y--
D		y++
L	x--	
R	x++	