

# TP3 Vasileios Skarleas et Yanis Sadoun

---

## Exercice 1

Voici le code pour cet exercice :

```
void generate_instances(int *debut, int *fin, int n, int Fmax){
    for (int i = 0; i < n; ++i)
    {
        debut[i] = nb_random(0, Fmax); //taking any number on the intervalle
        fin[i] = debut[i] + nb_random(1, Fmax - debut[i]); //starting from the
        beginning of the interval we add minimum 1 up to maximum - start time (in
        order to remain on the Fmax limit)
    }
}
```

L'agorithme est le suivant:

```
tant que i de 1 à n
    début[i] <- nombre_aléatoire(0 à max)
    fin[i] <- debut[i] + nombre_aléatoire(1, nouveau_max)
```

La condition de nouveau max est d'être inférieure ou égale à max.

La complexité de ce code peut être analysé comme suit :

La fonction génère des instances pour un problème d'ordonnancement en assignant aléatoirement des dates de début et de fin à n tâches, avec certaines contraintes.

- La boucle for itère n fois, où n est le nombre de tâches.
- `debut[i] = nb_random(0, Fmax)` génère un nombre aléatoire pour la date de début de la tâche. Cette opération est considérée comme  $O(1)$ , une opération en temps constant.
- `fin[i] = debut[i] + nb_random(1, Fmax - debut[i]);` : Génère un autre nombre aléatoire pour la durée de la tâche et l'ajouter à la date de début pour obtenir la date de fin. Cette opération est également  $O(1)$ .

Par conséquent, la complexité de `generate_instances` est  $O(n)$ , où n est le nombre de tâches à générer.

## Exercice 2

### Q1

La complexité du code de `calcule_optimale` mentioné sur le sujet du TP peut être analysée comme suit:

- On constate que pour chaque tâche j, l'algorithme parcourt toutes les tâches précédentes (de 0 à j-1) pour trouver la dernière tâche compatible. Le pire des cas est  $O(j)$ .

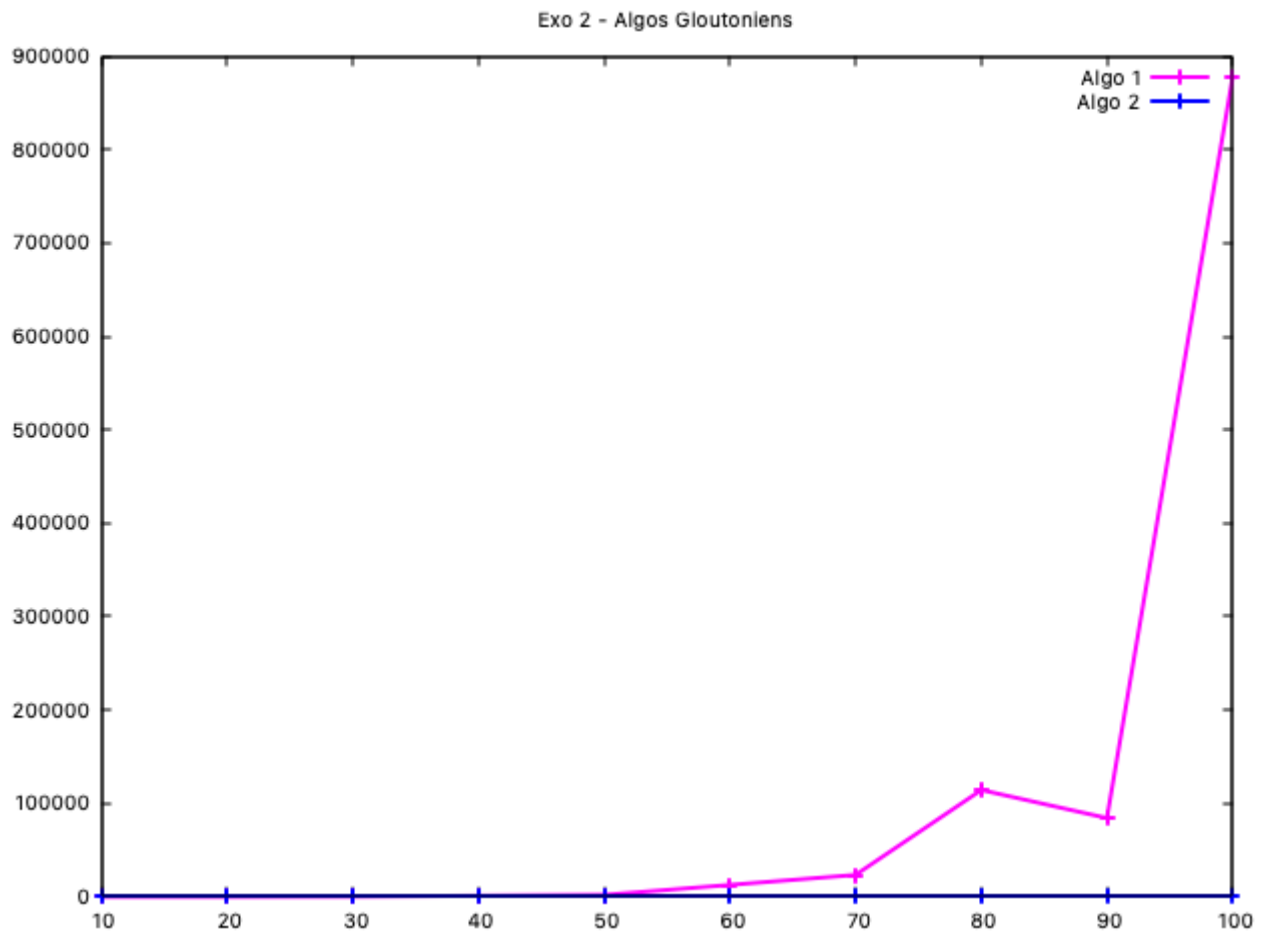
- L'algorithme fait deux appels récursifs : un pour le cas `der_j` et un autre pour le cas où la tâche `j` n'est pas incluse.

Par conséquent, la complexité de cet algorithme est exponentielle, plus précisément en  $O(2^n)$  où  $n$  est le nombre total de tâches.

Nota bene: En pratique, cet algorithme peut devenir très lent pour un nombre relativement faible de tâches, étant donné que le nombre total d'opérations augmente très rapidement à mesure que le nombre de tâches augmente. C'est d'ailleurs ce qui est visible sur le graphe ci-dessous.

```
int calcule_OPT(int *deb, int *fin, int j)
{
    if (j < 0) // 0(1)
    {
        return 0;
    }
    else
    {
        int der_j = -1; //0(1)
        // Trouver la dernière tâche compatible avec la tâche j
        for (int i = 0; i < j; i++) // j * 0(1) = 0(j)
        {
            if (is_compatible(i, j, deb, fin)) // 0(1)
            {
                der_j = i; // 0(1)
            }
        }

        // Calculer récursivement l'OPT en incluant ou excluant la tâche j
        return max(1 + calcule_OPT(deb, fin, der_j), calcule_OPT(deb, fin,
j - 1));
    }
}
```



## Q2

La complexité de l'algorithme `calcule_OPT_glouton` est  $O(n)$ . En effet, l'algorithme contient principalement une boucle `for` qui itère sur l'ensemble des tâches, allant de 0 à `nbTaches - 1`. À l'intérieur de cette boucle, toutes les opérations effectuées, sont en temps constant, c'est-à-dire  $O(1)$ . Puisque ces opérations en temps constant sont répétées pour chaque tâche, la complexité globale de l'algorithme est dominée par le nombre d'itérations de la boucle, qui est proportionnel au nombre de tâches `n`.

Dès lors, la complexité totale est en  $O(n)$  (linéaire par rapport à `n`).

```
int calcule_OPT_glouton(int *deb, int *fin, int nbTaches)
{
    int dispo = 0; // O(1)
    int k = 0; // O(1)

    for (int j = 0; j < nbTaches; j++) // n * 3*O(1) = O(n)
    {
        if (deb[j] >= dispo) // O(1)
        {
            k++; // O(1)
            dispo = fin[j]; // O(1)
        }
    }
}
```

```
    return k; // O(1)
} => O(total) = 3 * O(1) + O(n) = O(n)
```

### Q3

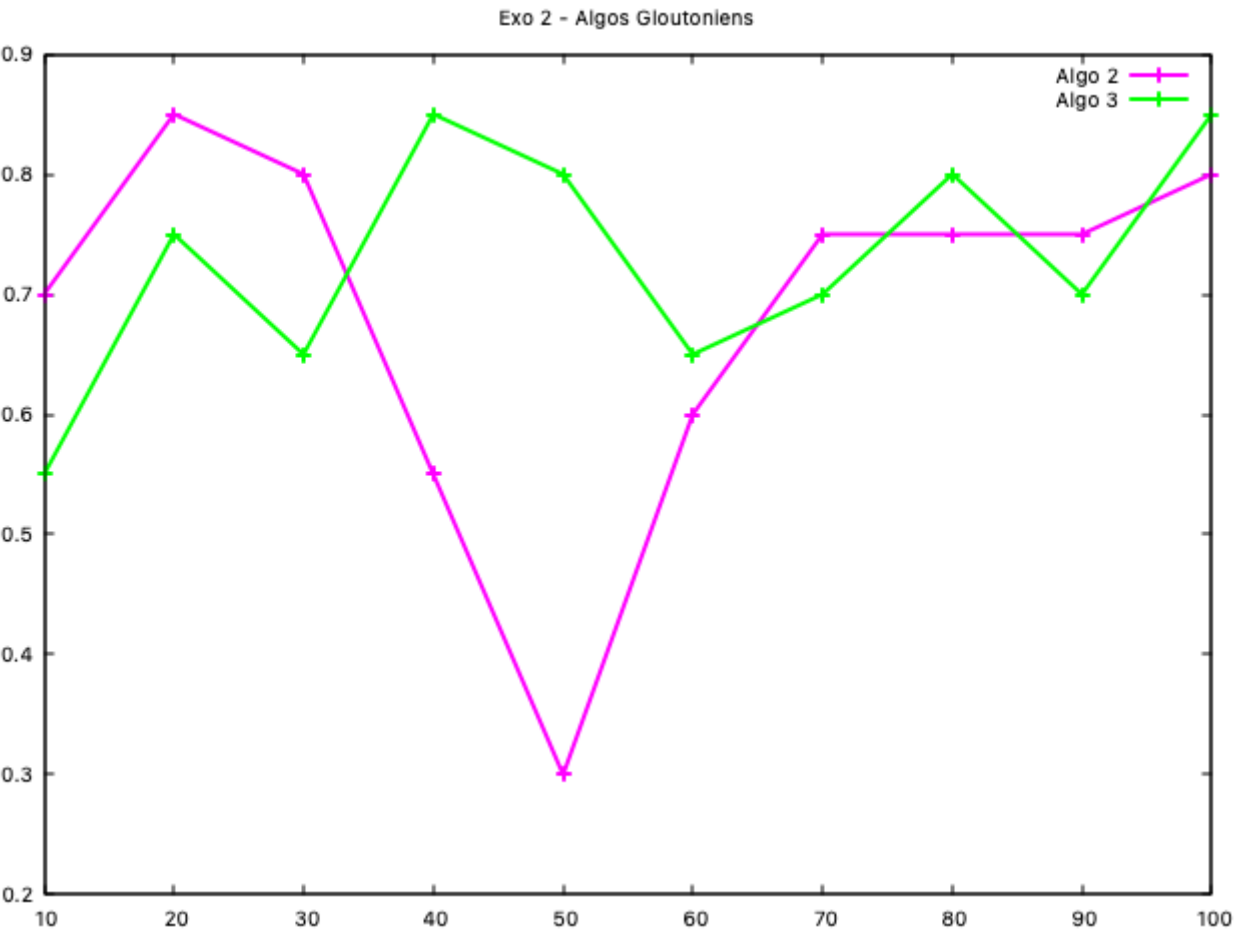
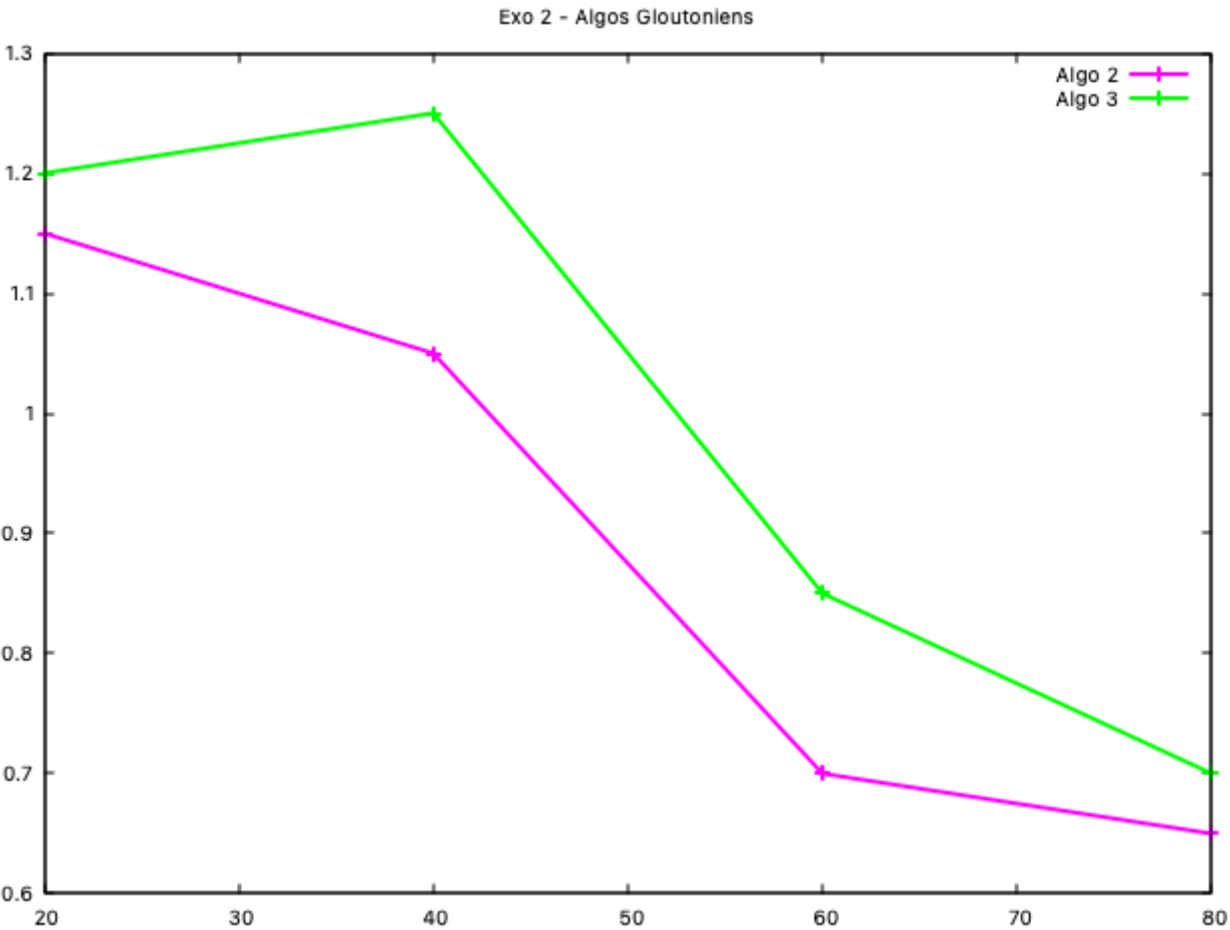
L'algorithme récursif de la question 1, montre une complexité qui croît rapidement avec la taille de l'entrée, comme en témoigne le tracé rose sur le graphe, qui indique une augmentation exponentielle du temps d'exécution. En revanche, l'Algorithme 2, qui est une approche gloutonne, présente une complexité linéaire  $O(n)$ , ce qui se traduit par un temps d'exécution beaucoup plus court, à peine visible sur le graphe et représenté en bleu. Ce contraste marque l'efficacité supérieure de l'algorithme glouton par rapport à l'approche récursive pour ce problème spécifique.

### Q4

L'algorithme `calcule_OPT_glouton2` enrichit la version gloutonne précédente en enregistrant explicitement les tâches sélectionnées pour l'ordonnancement. Il utilise pour cela un tableau statique `tacheSelected` de booléens où chaque élément correspond à une tâche : si `tacheSelected[j]` vaut `true`, cela signifie que la tâche `j` a été retenue dans l'ordonnancement ; si c'est `false`, la tâche `j` n'a pas été retenue car elle entre en conflit avec une tâche précédemment choisie. Bien que cette information supplémentaire soit sauvegardée, elle n'influe pas sur le processus décisionnel de l'algorithme glouton lui-même. La complexité de cet algorithme demeure linéaire,  $O(n)$ , car il parcourt une seule fois l'ensemble des tâches, indépendamment de l'ajout de cette trace.

Nota bene: Cette stratégie semble être analogue à une approche de programmation dynamique (vu au td4 du cours d'algorithmie), car elle stocke des informations supplémentaires qui pourraient être utiles pour des décisions ultérieures.

On obtient les graphiques suivants:



Ici, algo2 correspond à calcule\_OPT\_glouton et algo3 correspond à calcule\_OPT\_glouton2.

A travers ces graphes , on constate que bien que les deux algorithmes présentent la même complexité théorique, leurs performances pratiques diffèrent. Cela illustre comment une opération individuelle de complexité  $O(1)$  peut influencer, même subtilement, le comportement global d'un algorithme.

### Nota bene

La différence de performance entre les deux algorithmes n'est pas nettement visible sur le graphique, en raison du choix de l'échelle des tailles de séquence pour les tests. Les temps d'exécution pour la fonction de complexité  $O(n)$  ne sont pas clairement distingués car les tests n'ont pas été effectués avec une gamme suffisamment large de valeurs de  $n$ . Cela est d'autant plus notable que la même fonction d'appel est utilisée pour la question 1, qui a une complexité de  $O(2^n)$  et qui requiert un temps d'exécution considérablement plus long.

## Exercice 3

### Q1

Pour optimiser l'ordonnancement et minimiser le nombre de machines nécessaires, il est important de déterminer la séquence de tri des tâches la plus efficace. Parmi les trois stratégies de tri proposées, trier les tâches par ordre croissant de leurs dates de début  $d[j]$  s'avère être l'approche optimale. Cette méthode priorise les tâches selon leur moment de commencement, facilitant une utilisation des machines plus stratégique et plus efficace.

En examinant les contre-exemples pour les deux autres méthodes de tri, on peut démontrer leur inefficacité :

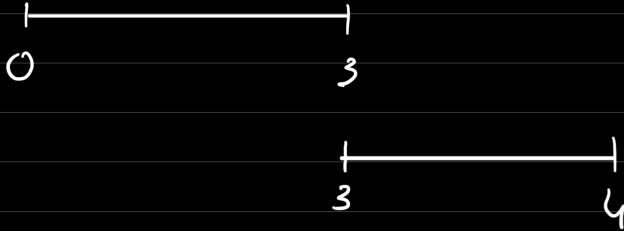
#### **Trier les tâches en ordre croissant de $f[j] - d[j]$ :**

Cette methode (la durée de la tâche) ne prend pas en compte les chevauchements potentiels qui peuvent survenir avec des tâches plus longues commençant plus tôt.

17:51 Vendredi 23 février

- tâche 1 commence à 0 → finit à 3

- tâche 2 commence à 3 → finit à 4



⇒ ordre : tâche 2 puis tâche 1

Ici la machine 1 est libre à 4, donc on a besoin d'une deuxième machine pour effectuer la 1<sup>ère</sup> tâche

Or avec trier par ordre croissant ⇒ utilisation d'une machine

### Trier les tâches en ordre croissant de $f[j]$

Cette méthode peut conduire à un intervalle de temps inutilisé si une tâche ayant une date de début tardive se termine avant d'autres tâches ayant commencé plus tôt.

17:58 Vendredi 23 février

Apple Pencil 100 %

Considérons 4 tâches :

Task 1: [0, 5]  
Task 2: [1, 6]  
Task 3: [7, 8]  
Task 4: [5, 9]

En considérant un tri par ordre croissant des  $f[j]$ ,  
3 machines seront nécessaires :

machine 1 : 1, 3  
machine 2 : 2  
machine 3 : 4

• Or en utilisant tri des tâches en ordre croissant des  $d[j]$  : 2 machines nécessaires

## Algorithme

Dès lors, voici le code de notre algorithme qui trie les tâches par ordre croissant des  $d[j]$ :

```
void init_servers(int n, int *tab)
{
    for (int i = 0; i < n; i++)
    {
        tab[i] = -1;
    }
    return;
}

int glouton(int *deb, int *fin, int nbTaches)
{
    int p = 0; // servers working currently
    int servers[nbTaches]; // maximum number of machines is the number of requests

    init_servers(nbTaches, servers);

    for (int j = 0; j < nbTaches; j++)
    {
        int i = 0;
```



```

    bool need_new_server = true;
    while (i < p && need_new_server) // we test every server that is
working if it has finished the task or not
    {
        {
            if (servers[i] <= deb[j])
            {
                servers[i] = fin[j];
                need_new_server = false;
            }
            i++;
        }
    }

    if (need_new_server) // every server is working. We add a new one
on the network
    {
        servers[p] = fin[j];
        p++;
    }
}

return p;
}

```

## Complexité

Sa complexité peut être analysée comme suit :

- La boucle dans cette fonction parcourt le tableau `servers` de taille `n` une seule fois pour initialiser chaque élément à -1. La complexité est donc  $O(n)$ .
- Il y a une boucle `for` qui parcourt toutes les tâches, donc elle s'exécute `n` fois.
- À l'intérieur de la boucle `for`, il y a une boucle `while` qui parcourt les serveurs existants. Dans le pire des cas, si chaque tâche nécessite une nouvelle machine, cette boucle s'exécute `p` fois, où `p` est le nombre de machines déjà utilisées. Dans le pire des cas, `p` peut être aussi grand que `n`.
- La vérification de la disponibilité du serveur (`if (servers[i] <= deb[j])`) est une opération en temps constant, en  $O(1)$  mais comme elle se trouve à l'intérieur de la boucle imbriquée, cette opération pourrait être exécutée jusqu'à `n` fois dans le pire des cas.

Dès lors, la complexité totale de l'algorithme est en  $O(n^2)$  dans le pire des cas (c'est quadratique).

## Q2

### Algorithme

Pour calculer cette profondeur, une première étape consiste à trier les tableaux de début et de fin d'intervalles en ordre croissant. Cela permet d'établir une condition nécessaire : `nb_machines_besoin >= profondeur`. Une fois les intervalles triés, l'algorithme de calcul de la profondeur est appliqué pour déterminer ce nombre maximal de chevauchements, comme suit :

```
tri_selon_la_fin(deb, nbTaches)
tri_selon_la_fin(fin, nbTaches)
depth <- 0 //Initialisation de la profondeur à zéro

tant que i < nbTaches & j < nbTaches
  si deb[i] < fin[j] alors
    p <- p + 1
    i <- i + 1
  sinon
    j <- j + 1
  fin si

  depth <- max(p, depth)
fin tant que
retourner depth
```

## Complexité

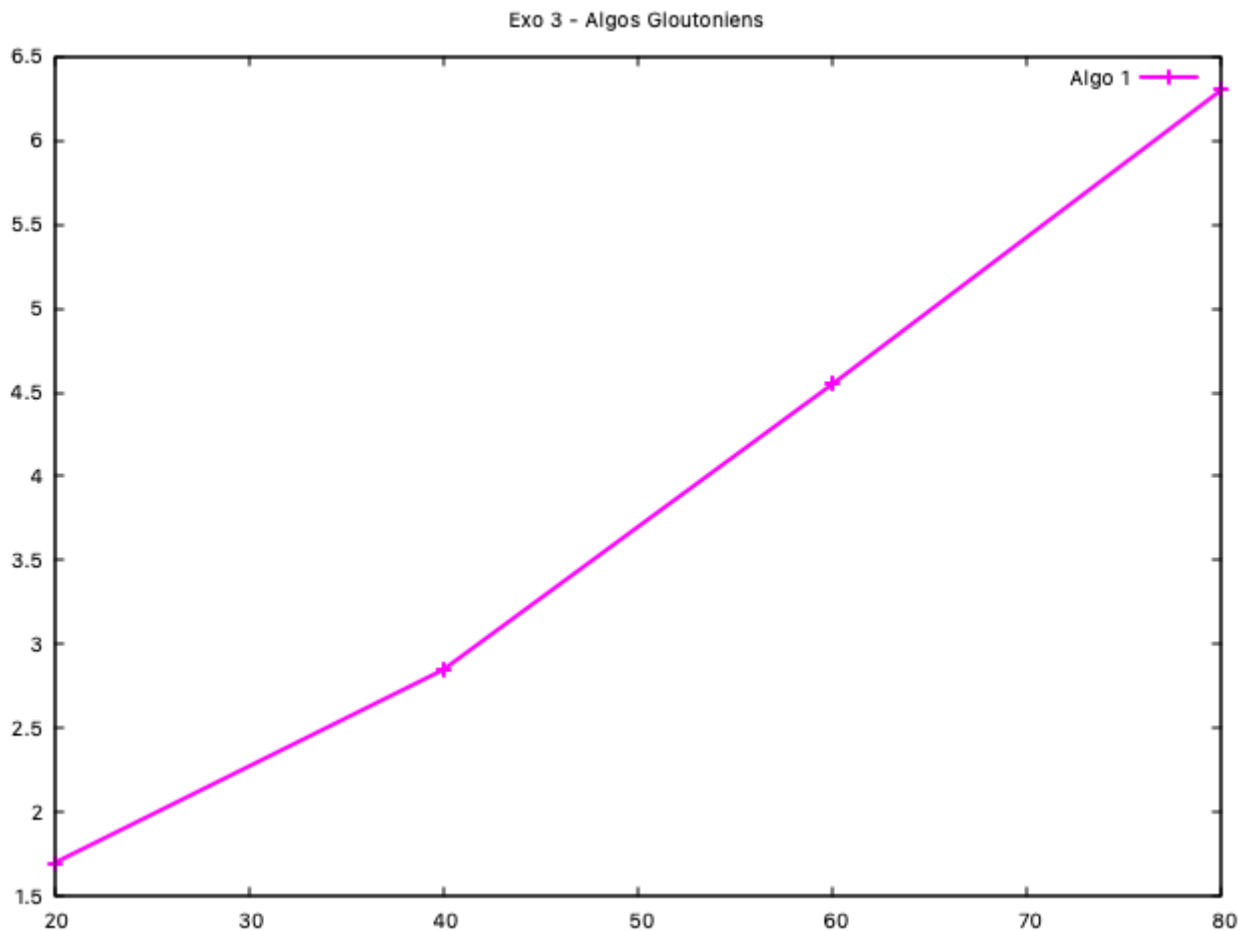
La complexité de cet algorithme peut être analysée comme suit :

- `tri_selon_la_fin(deb, nbTaches)` et `tri_selon_la_fin(fin, nbTaches)` qui utilise le tri fusion. La complexité de chaque tri est  $O(n \log n)$ , où  $n$  est `nbTaches`. Puisque le tri est effectué deux fois, la complexité totale pour cette étape est  $2 * O(n \log n) = O(n \log n)$ .
- La boucle s'exécute tant que `i` et `j` sont inférieurs à `nbTaches`. En pratique, cela signifie qu'elle s'exécutera pour chaque extrémité d'intervalle (début et fin), soit un total de  $2n$  fois dans le pire des cas (chaque `i` et `j` va de 0 à `nbTaches - 1`). Les opérations à l'intérieur de la boucle sont toutes des opérations en temps constant,  $O(1)$ .

En combinant ces éléments, la complexité totale de l'algorithme est :

- Pour les tris :  $O(n \log n)$ .
- Pour la boucle :  $O(2n) = O(n)$ .

Par conséquent, la complexité globale de l'algorithme est dominée par l'étape de tri, soit  $O(n \log n)$ . Cette complexité est visible sur la graphique suivant:



### Q3

Nous avons réalisé des tests expérimentaux sur deux ensembles d'intervalles pour vérifier l'hypothèse selon laquelle le nombre de machines requis dans un ordonnancement optimal correspond à la profondeur de l'ensemble des intervalles. Voici les ensembles testés :

- Pour le premier ensemble :
  - Les dates de début sont [1, 2, 4, 6, 8].
  - Les dates de fin sont [3, 5, 7, 9, 10].
- Pour le second ensemble :
  - Les dates de début sont [1, 3, 0, 5, 8, 5].
  - Les dates de fin sont [2, 4, 6, 7, 9, 9].

Les résultats des tests sont cohérents avec l'hypothèse : le premier ensemble nécessite 2 machines, tandis que le second en nécessite 3. Ces résultats correspondent à la profondeur maximale des intervalles dans chaque ensemble, confirmant ainsi que le nombre de machines dans un ordonnancement optimal est bien égal à la profondeur de l'ensemble d'intervalles des tâches.

## Exercice 4

### V1

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

```
int compare(const void *a, const void *b) {
    return *(int *)a - *(int *)b;
}

int main() {
    int N; // Nombre de Pilipius
    scanf("%d", &N);
    long long C; // Coût du cadeau
    scanf("%lld", &C);

    int budgets[N]; // Tableau pour stocker les budgets de chaque Pilipiu
    long long sum_of_budgets = 0; // Somme totale des budgets

    for (int i = 0; i < N; i++) {
        scanf("%d", &budgets[i]);
        sum_of_budgets += budgets[i];
    }

    if (sum_of_budgets < C) {
        printf("IMPOSSIBLE\n");
        return 0;
    }

    qsort(budgets, N, sizeof(int), compare); // Trier les budgets

    int contributions[N];
    long long remaining_cost = C;

    for (int i = 0; i < N; i++) {
        // Contribution calculée comme la part équitable ou le budget
        // maximum, selon le plus petit
        long long equitable_share = remaining_cost / (N - i);
        contributions[i] = (budgets[i] < equitable_share) ? budgets[i] :
equitable_share;
        remaining_cost -= contributions[i];
    }

    // Affichage des contributions
    for (int i = 0; i < N; i++) {
        printf("%d\n", contributions[i]);
    }

    return 0;
}
```

Ce code suit les étapes suivantes :

- Lire le nombre de Pilipius et le coût du cadeau.
- Lire les budgets de chaque Pilipiu, les additionner et les trier.
- Si la somme des budgets est inférieure au coût du cadeau, afficher "IMPOSSIBLE".

- Sinon, répartir le coût du cadeau parmi les Pilipius en commençant par le budget le plus bas, tout en s'assurant que chaque contribution est le plus petit entre le budget du Pilipiu et sa part équitable du coût restant.
- Afficher les contributions de chaque Pilipiu.

Cette approche gloutonne vise à minimiser la contribution la plus élevée en attribuant d'abord le montant que chaque Pilipiu peut raisonnablement contribuer, compte tenu du budget restant et du nombre de Pilipius restants.

L'algorithme glouton que nous proposons pour le problème des Pilipius n'est pas nécessairement correct pour toutes les instances possibles. En effet, le problème réside dans la manière dont l'algorithme répartit le coût du cadeau parmi les Pilipius. Il tente de minimiser la contribution la plus élevée en attribuant d'abord la contribution que chaque Pilipiu peut raisonnablement payer, compte tenu de son budget et du coût total restant. Cependant, cette approche ne garantit pas toujours que la solution trouvée soit optimale selon les critères énoncés, notamment : La plus grande contribution est minimale. Si plusieurs solutions optimales sont possibles, choisir celle où la deuxième plus grande contribution est minimale, et ainsi de suite.

## V2

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

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

int sum_of(int n, int *tab)
{
    int sumall = 0;
    for (int i=0; i < n; i++){
        sumall = tab[i] + sumall;
    }
    return sumall;
}

int compare(const void *a, const void *b) { //difference to decide which
is greater
    return *(int *)a - *(int *)b;
}

int main()
{
    int N;
    scanf("%d", &N);
    int C;
    scanf("%d", &C);
    int amounts[N];
    for (int i = 0; i < N; i++) {
```

```
        int B;
        scanf("%d", &B);
        amounts[i] = B;
    }

    // Write an answer using printf(). DON'T FORGET THE TRAILING \n
    // To debug: fprintf(stderr, "Debug messages...\n");

    if (sum_of(N, amounts) < C)
    {
        printf("IMPOSSIBLE\n");
    }
    else
    {
        qsort(amounts, N, sizeof(int), compare); // Trier les budgets
        int remains = C;
        int players = N;
        for (int i = 0; i < N; i++)
        {
            if (remains/players > amounts[i])
            {
                printf("%d\n", amounts[i]);
                remains = remains - amounts[i];
                players--;
            }
            else
            {
                printf("%d\n", remains/players);
                remains = remains - remains/players;
                players--;
            }
        }
    }

    return 0;
}
```

Ce code suit les étapes suivantes :

1. Trier le tableau des budgets
2. Décider localement selon le montant restant si l'utilisateur va payer une partie ou le maximum de son budget.
3. L'algorithme prend en compte le nombre de participants ayant déjà contribué pour ajuster les contributions ultérieures.