

# [CYBER1][2024-2025] CORRECTION Rattrap (2h00)

## Algorithmique 1

NOM : \_\_\_\_\_

PRÉNOM : \_\_\_\_\_

Vous devez respecter les consignes suivantes, sous peine de 0 :

- I) Lisez le sujet en entier avec attention
- II) Répondez sur le sujet
- III) Ne trichez pas
- IV) Ne détachez pas les agrafes du sujet
- V) Écrivez lisiblement vos réponses (si nécessaire en majuscules)
- VI) Vous devez écrire les algorithmes et structures en langage C (donc pas de Python ou autre)

## 1 Bases d'Algorithmique (10 points)

### Question

- 1.1 (2 points) Exécutez l'algorithme suivant, et notez l'état d'avancement des variables (inscrivez l'état initial dans la ligne prévue à cet effet) :

Vous exécuterez la fonction suivante avec comme paramètres  $a = 2110$  et  $b = 27972$  :

```
int FunctionXYZ(int a, int b)
{
    int num = 0;

    if ((a > 0) && (b < 0))
        return (-1);

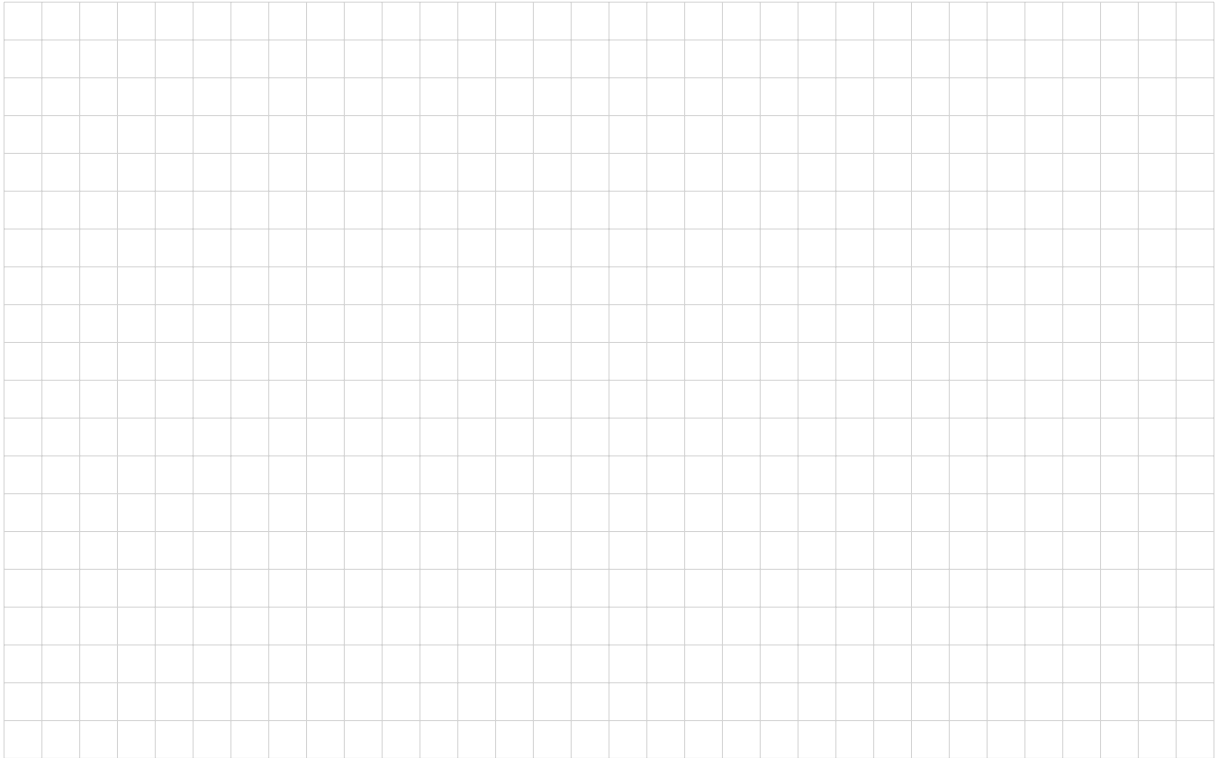
    while ((a > 0) && (b > 0))
    {
        if ((num % 2) == 0)
            num += 1;
        else
            num *= 2;
        a = a - 100;
        b = b / 10;
    }
    return (num);
}
```

tour	a	b	num
<i>État initial</i>	2110	27972	0
1	2010	2797	1
2	1910	279	2
3	1810	27	3
4	1710	2	6
5	1610	0	7

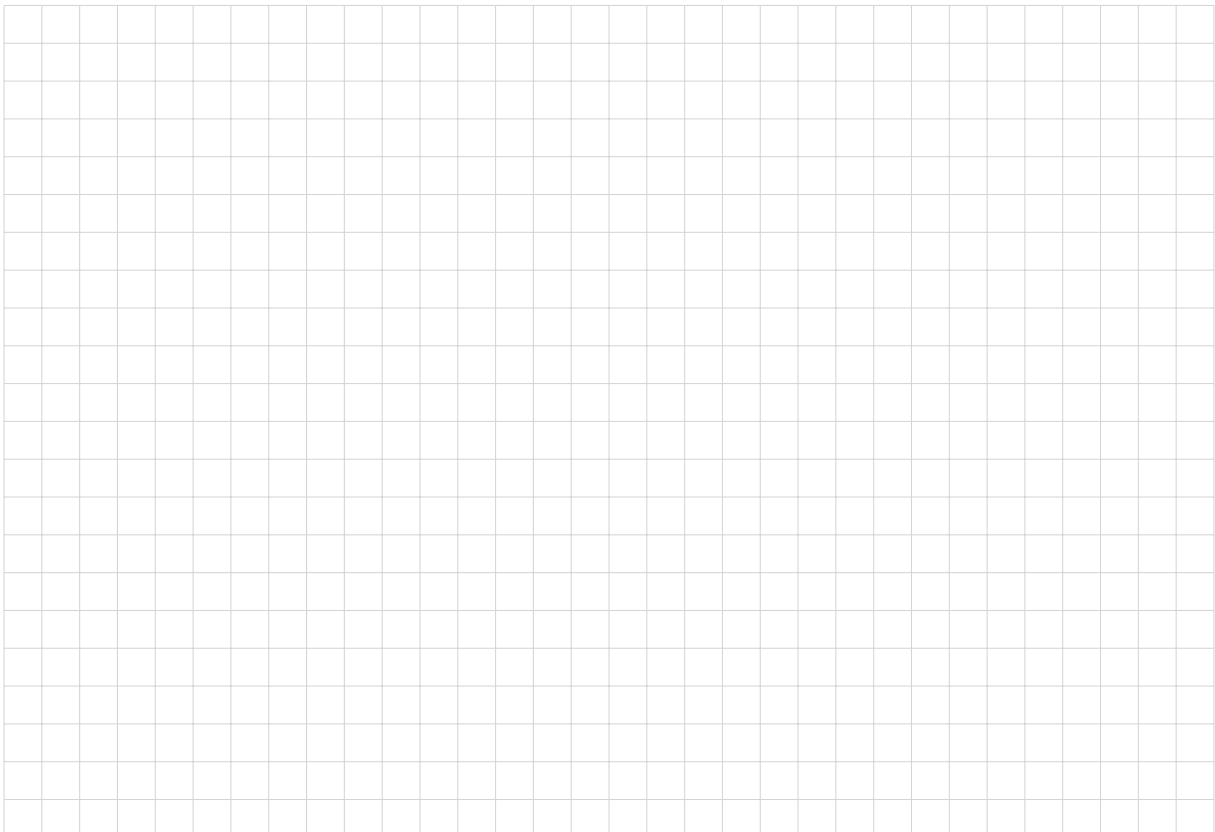
## Algorithmes

- 1.2 (2 points) Écrivez une fonction récursive « *NbChiffres* » calculant la longueur d'un nombre entier (c'est-à-dire le nombre de chiffres qui le constituent)

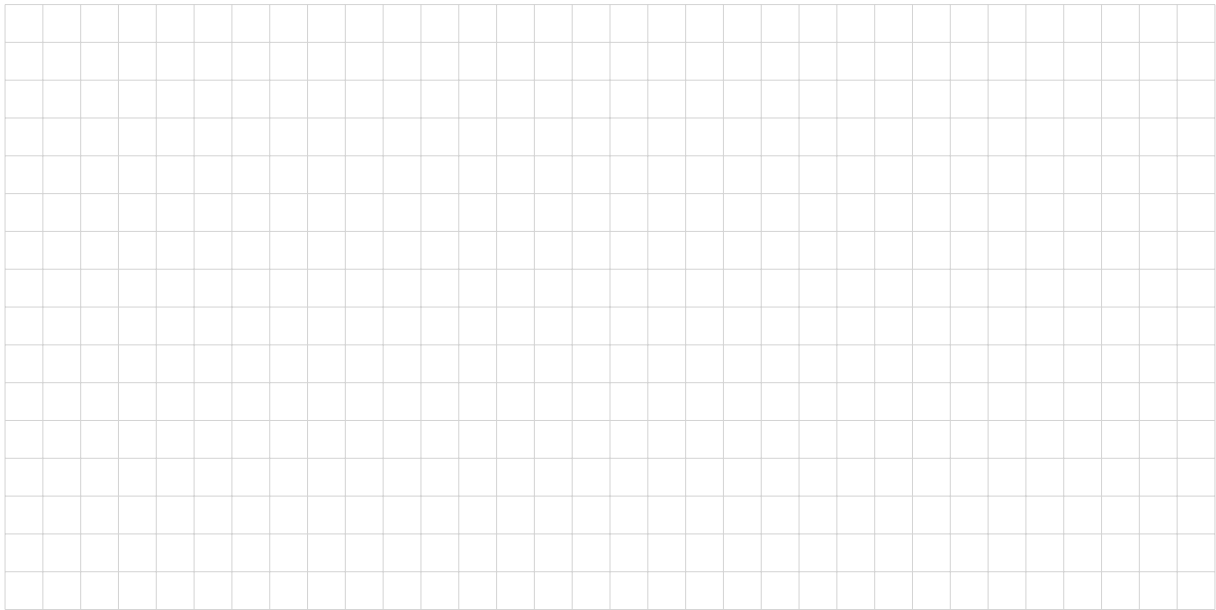
*Le nombre « 0 » est de longueur 1*



- 1.3 (2 points) Écrivez une fonction itérative « *my\_strlen* » calculant la taille d'une chaîne de caractères



- 1.4 (2 points) Écrivez une fonction « *Count\_Odd* » renvoyant la quantité de nombres impairs contenus dans un tableau d'entiers



- 1.5 (2 points) Indiquez et corrigez la ligne incorrecte dans ce tri à bulles (on considère que la fonction « swap » est disponible)

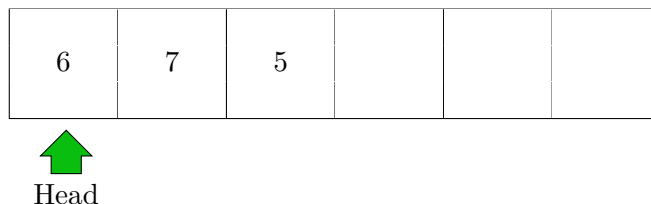
```
void BubbleSort(int tab[], int len)
{
    for (int i = (len - 1); i > 0; i--)
    {
        for (int j = 0; j < i; j++)
        {
            %%% if (tab[j] > tab[i]) %%% ERREUR ICI
            % if (tab[j] > tab[j + 1]) % <= CORRECTION
            {
                swap(tab, len, j, (j + 1));
            }
        }
    }
}
```

## 2 Listes/Piles/Files (7 points)

### Questions

**2.1 (1 point)** Écrivez l'état d'une file après avoir effectué ces opérations (la file est considérée comme initialement vide), puis, indiquez quel élément sortira de la file lors du prochain « dequeue », ainsi que celui qui sortira en dernier :

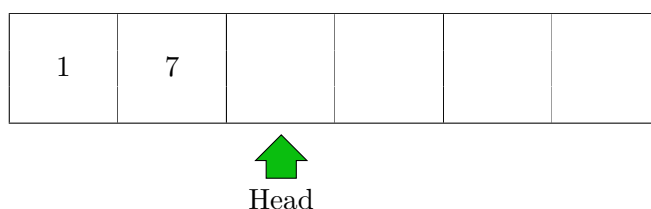
enfiler 3, enfiler 1, enfiler 4, défiler, enfiler 2, défiler, défiler, enfiler 6, enfiler 7, défiler, enfiler 5



Prochain élément qui sortira :            6                      Dernier élément qui sortira :            5

**2.2 (1 point)** Écrivez l'état d'une pile après avoir effectué ces opérations (la pile est considérée comme initialement vide), puis, indiquez quel élément sortira de la pile lors du prochain « pop », ainsi que celui qui sortira en dernier :

empiler 5, empiler 3, empiler 2, dépiler, dépiler, empiler 4, dépiler, dépiler, empiler 1, empiler 6, dépiler, empiler 7



Prochain élément qui sortira :            7                      Dernier élément qui sortira :            1

**2.3 (0,5 point)** Déduisez maintenant la caractéristique de chacun des conteneurs :

Ce conteneur maintient en sortie l'ordre dans lequel les éléments sont insérés	FILE
--	------

Ce conteneur modifie en sortie l'ordre dans lequel les éléments sont insérés	PILE
--	------

**2.4 (1,5 point)** En admettant que l'on dispose d'une pile vide et que les éléments « 1 2 3 4 5 6 » arrivent en entrée dans cet ordre exclusivement, décrivez les scénarios permettant d'obtenir les sorties suivantes. Si c'est impossible, indiquez simplement « impossible » :

*exemple : pour « A B C » en entrée, on peut obtenir « B C A » en sortie en faisant :*

*« push A », « push B », « pop », « push C », « pop », « pop »*

*On a bien inséré A, puis B, puis C, mais l'ordre de sortie est différent suivant les « pop »*

3, 4, 5, 2, 6, 1

push 1, push 2, push 3, pop, push 4, pop, push 5, pop, pop, push 6, pop, pop

1, 3, 5, 2, 4, 6

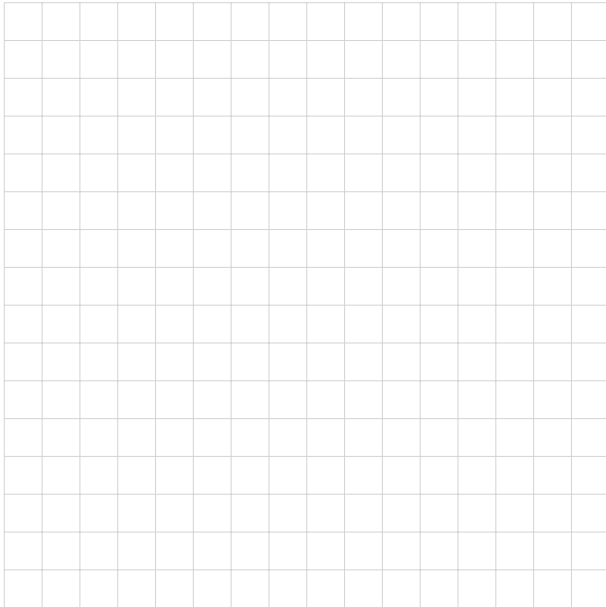
impossible

2, 5, 4, 6, 3, 1

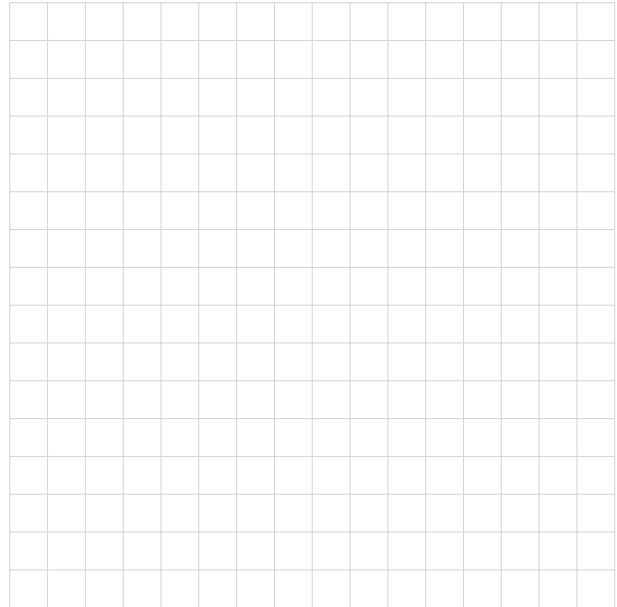
push 1, push 2, pop, push 3, push 4, push 5, pop, pop, push 6, pop, pop, pop

## Algorithmes

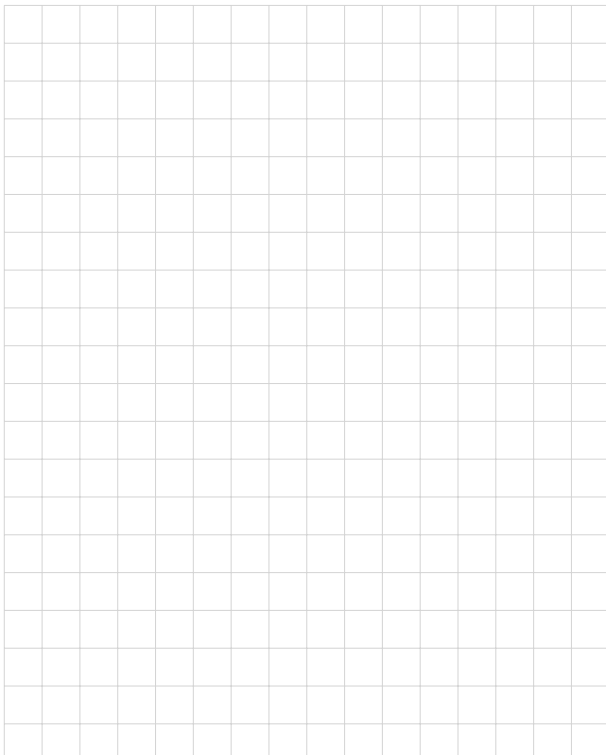
2.5 (0,5 point) Écrivez une structure de données « *my\_stack* » pouvant servir de pile (à base de pointeurs ou de tableaux)



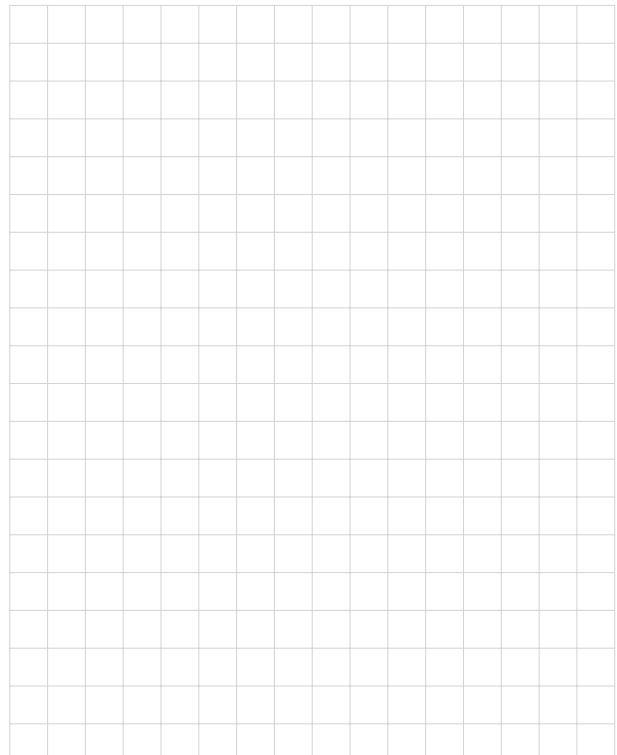
2.6 (0,5 point) Écrivez une structure de données « *my\_queue* » pouvant servir de file (à base de pointeurs ou de tableaux)



2.7 (1 point) Écrivez une fonction « *push* » pouvant servir à empiler un élément dans votre précédente structure « *my\_stack* »



2.8 (1 point) Écrivez une fonction « *pop* » pouvant servir à dépiler un élément dans votre précédente structure « *my\_stack* »



### 3 Problème : Ordonnancement (3 points)

L'ordonnanceur d'un système d'exploitation est un composant distribuant du temps d'exécution à chaque tâche selon divers critères dont la priorité. Dans le problème traité ici, seuls deux critères nous intéressent : la *priorité* du processus, et la *durée d'exécution* totale déjà réalisée.

Dans un ordonnanceur, les nouveaux processus sont ajoutés au fur et à mesure dans la file d'attente, mais, on n'attribue du temps qu'à celui qui est en tête de la file. L'insertion dans la file d'attente n'est pas une insertion classique en queue, mais plutôt n'importe où dans la file selon des critères.

Ici, le critère le plus important sera la *priorité* : plus la valeur de priorité est basse, moins le processus devra attendre pour être exécuté (il sera donc très prioritaire). Exemple : un processus avec une priorité de 2 attendra moins qu'un processus avec une priorité de 5. La valeur de priorité est un entier positif (0 étant la plus petite valeur, et donc, la priorité maximale).

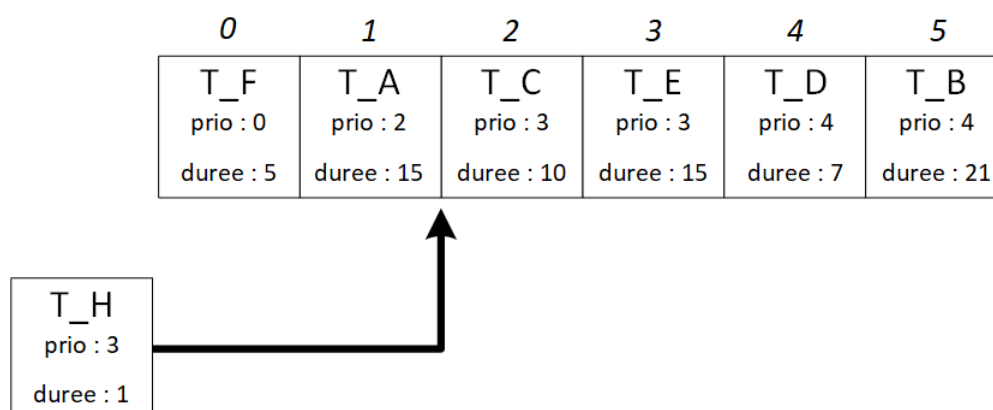
Le deuxième critère est la *durée d'exécution* totale déjà réalisée : plus un processus va prendre de temps d'exécution, moins il sera prioritaire (les tâches récentes passeront avant les plus anciennes). Exemple : un processus avec une priorité de 4 ayant été exécuté seulement 7 minutes attendra moins qu'un processus avec une priorité de 4 ayant été exécuté 21 minutes, mais, il attendra toujours plus qu'un processus de priorité 1 ayant été exécuté 42 minutes.

Enfin, si une tâche déjà présente dans la file d'attente a les mêmes caractéristiques, la nouvelle tâche passera après (en d'autres termes : si deux processus ont les mêmes priorités et temps d'exécution, alors le premier arrivé est le premier servi).

Pour résoudre ce problème, vous avez accès à cette structure **struct task\_struct** décrivant une tâche/un processus :

```
struct task_struct
{
    void *stack;
    int PID;    // Identifiant du processus
    int prio;   // Priorite
    int usage;  // Temps total d'execution
};
```

Dans ce problème, il faut donc trouver l'emplacement idéal dans un **struct task\_struct\*\***, c'est-à-dire un tableau de **struct task\_struct\***. La taille du tableau est maintenue dans un paramètre. L'exemple suivant illustre une tâche *T\_H* de priorité 3, et de durée totale 1 minute, insérée avant la tâche *T\_C* de priorité 3 et de durée totale 10 minutes, mais après la tâche *T\_A* de priorité 2 et de durée totale 15 minutes. La file d'attente est toujours triée dans le bon ordre, et elle doit donc le rester grâce à vos fonctions.



Vous avez également le droit d'utiliser la fonction **InsertList** prenant ces paramètres :

**InsertList(struct task\_struct \*\*L, struct task\_struct \*elt, int pos)**

- *struct task\_struct \*\*L* : le conteneur où insérer le nouvel élément (ici la file d'attente)
- *struct task\_struct \*elt* : l'élément à insérer (ici la structure concernant le processus)
- *int pos* : la position dans le conteneur (0 étant la tête/le prochain processus à être exécuté)

**3.1 (2 points) Écrivez d'abord une fonction parcourant la file d'attente pour trouver la position du nouveau processus selon les critères précédents :**

```
int CalculatePosition(struct task_struct **sched_list,
                    int sched_list_len,
                    struct task_struct *new_task)
```



3.2 (1 point) Écrivez ensuite une fonction insérant le processus à sa place et renvoyant la nouvelle taille de la liste :

```
int InsertProcess(struct task_struct **sched_list,  
                 int sched_list_len,  
                 struct task_struct *new_task)
```

# **SUJET RATRAPAGE**

## **ALGORITHMIQUE 1**