1er pas d'ALGORITHMIQUE

A - DÉFINIR : Un **algorithme** est une suite finie et non ambiguë d'opérations ou d'instructions permettant de résoudre une classe de problèmes (du nom du mathématicien perse Al-Khwârizmî vers 780)

Ainsi, cette notion est très adaptée à l'informatique (l'ordinateur étant le réceptacle de ces instructions)

B- RÉDACTION : Il peut être écrit en langage « naturel » ou en pseudo code et **implémenté (**=traduit) dans différents langages : Python,Javascript, calculatrice...

Quelques exemples d'instructions :

Langage « naturel »	Pseudo code	Python	Javascript
Affecter à A la valeur 5	A ← 5	A=5	var A=5
Afficher A	Afficher A	print(A)	alert(A) / console.log(A)
Saisir A	Saisir A	A=float(input ('A='))	Prompt A

Si *Condition* **Alors faire**
Instructions1
Sinon faire
Instructions2
	Si *Condition* **Alors**
	Instructions1
	Sinon
	Instructions2
	FinSi
		if *Condition* **:**
		Instruction1
		else :
		Instruction2
			if (condition1) {
			instructions 1; }
			else {
			instructions2;}

Pour i allant de 3 à 7 par pas de 1 **faire** *Instructions*	**Pour** i allant de 3 à 7 *Instructions* **FinPour**	***for*** *i in range (3,8) :* *Instructions*	**for** (var i=3 ; i<8 ; i++) { instructions ; }
Tant que la condition est vraie **faire** *Instructions*	**Tant que** Condition est vraie **faire** *Instructions* ***Fin du Tant que***	**while** *condition :* *Instructions*	**while** (condition) { Instructions} ;

Citez quelques différences et points communs de ces langages :

.....

.....

.....

C- COMPLEXITÉ TEMPORELLE

En informatique, la question de performance est centrale (surtout si ce temps utile pour obtenir un résultat est « long »). De manière générale, le traitement de données dépend du volume de ces données et de la nature du traitement que l'on fait.

Par exemple, une boucle peut posséder un grand nombre de répétitions
le programme affiche …….. lignes

```
n=100
for i in range (n):
    print (i)
```

mais l'effet est encore démultiplié avec des boucles imbriquées.
le programme affiche …….. lignes

```
n=100
for i in range (n):
    for j in range (n):
        print(i,j)
```

Ce nouveau programme affiche …….. lignes.

```
n=100
for i in range (n):
    for j in range (n):
        for k in range(n):
            print(i,j,k)
```

On multiplie maintenant n par 10, donner le coefficient de multiplication du nombre de lignes dans chacun de ces 3 programmes.

.....

On parle alors de complexité TEMPORELLE de l'algorithme : on s'attache rarement à la détermination exacte de ce temps (on conserve une estimation) Remarque : bien que ce cours se limite à cette complexité temporelle, il existe aussi une complexité spatiale qui rend compte de l'espace mémoire pendant l'exécution

Pour mesurer la complexité temporelle,
nous avons déjà vu quelques instructions Python de mesure du temps.

```
from time import time
debut = time()
# Code dont on mesure le temps
fin = time()
print("Temps passé : ",fin - debut)
```

Évaluez la durée des 3 programmes ci dessus.
Comparez avec vos camarades

Le deuxième programme a mis ……. fois plus de temps pour s'exécuter

par contre le troisième,lui, a mis …….. plus de temps !!!

D - VALIDITÉ : CORRECTION et TERMINAISON d'un algorithme

Être certain qu'un algorithme donne le bon résultat est essentiel !!

1°) La **terminaison** stipule que les calculs décrits par l'algorithme s'arrêteront. Évidemment, savoir déceler cela avant toute implémentation évite le « plantage » de la machine.

On utilise souvent un **variant de boucle** : c'est une fonction entière, positive, qui décroit strictement

Il vérifie que la boucle se termine: f(i)=0 => !B (non B, ou B est un booléen)

2°) Afin de déterminer la **correction** d'un algorithme, on détermine un **invariant de boucle** : c'est une propriété ou une formule logique qui doit être vraie, à l'initialisation et à chaque itération de la boucle quel que soit le nombre d'itération. La difficulté réside donc à le trouver.

Exemple : dans un tableau de n entiers, trouver le plus grand élément.

```
def maximum(liste):              # recherche le maximum dans la liste
    n = len(liste)               # longueur de la liste
    i=0                          # le premier indice
    maxi = liste[0]              # on démarre au premier élément de la liste
    while i < n:                 # on parcourt la liste (i indice d'un élément de la liste)
        if liste[i]>maxi:        # la valeur rencontrée est supérieur à maxi
            maxi = liste[i]      # on affecte une nouvelle valeur à maxi
        i=i+1                    # on s'occupe de l'indice suivant
    return maxi                  # la liste est parcourue : maxi contient le maximum
```

variant de boucle : $f(i) = n - i$

invariants de boucle :

* liste (elle n'est jamais modifiée)
* maxi est le plus grand élément de la partie de la liste dont les indices varient entre 0 et i (à la fin de la ligne maxi = …)
* $0 \leq i \leq n$

Compléter l'algorithme suivant

```
q=0
n=      # a compléter
while n!=0:
    n-=3
    q+=1
print(q)
```

n =…… pour qu'il termine. n =…..…pour qu'il ne termine pas.

Déterminer la condition de terminaison sur le variant de boucle n :

Donner un invariant de boucle pour la fonction suivante qui calcule x à la puissance n :

……..
……..

```
def puissance(x,n):
    r=1
    for i in range(n):
        r=r*x
    return r
```

E- ALGORITHMES DE RECHERCHE

ALGORITHMES DE TRI

Trier un tableau de nombres, c'est ranger ces nombres dans l'ordre croissant.
Par exemple, le tableau [6 3 7 2 3 5] devient une fois trié [2 3 3 5 6 7] .
Bien sûr, on a souvent besoin de trier d'autre choses que des nombres (comme des mots par ordre alphabétique, des fichiers par longueur, des messages par date, les résultats d'une recherche par pertinence, des articles par référence, des personnes par date de naissance...), mais les algorithmes utilisés sont les mêmes.
Contrairement à ce qui se passe en général dans la vie courante, en informatique, on trie de très grandes quantités de données (quelques centaines de milliers d'éléments) à tout bout de champ. C'est pourquoi les informaticiens ont inventé de très nombreuses méthodes de tri, plus ou moins rapides et efficaces. Les ordinateurs passent énormément de temps à faire des tris : on considère aujourd'hui que les algorithmes de tri sont ceux qui sont les plus utilisés par les ordinateurs du monde entier !

Fiche d'identité de l'algorithme de **RECHERCHE SÉQUENTIELLE**

Principe : on cherche v dans une liste : on ……
……. → renvoie VRAI si v est trouvé, FAUX sinon

Exemple : feuilleter un livre page après page depuis le début pour retrouver une information

Pseudo Code

……..
……..
……..
……..
……..
……..

Python

```
def chercher1(v,list):          #recherche la présence de l'entier v dans la liste "list"
    for i in list:              # on parcourt la liste
        if i==v:                # v est trouvé dans la liste
            return True         # on indique qu'on a trouvé l'élément
    return False                # la liste est parcourue : v ne lui appartient pas

def chercher2(v,list):          #recherche la présence de l'entier v dans la liste "list"
    if v in list:               # on parcourt la liste
        return True             # v est trouvé dans la liste et on indique qu'on a trouvé l'élément
    return False                # la liste est parcourue : v ne lui appartient pas
```

TERMINAISON de l'algorithme

Comme prouver ici que l'algorithme remplit bien son rôle (Correction de l'algorithme) est « trivial » (mais ce ne sera pas le cas des algorithmes de tri!), nous allons concentrer nos efforts pour prouver que cet algorithme s'arrête.

Pour prouver qu'un algorithme s'arrête, on choisit une variable et on vérifie que la suite formée par les valeurs de cette variable au cours des itérations converge en un nombre fini d'étapes vers une valeur satisfaisant la condition d'arrêt. Cette variable s'appelle un **variant de boucle.**

Dans ce cas précis, on peut choisir la suite du nombre d'éléments $(n_i)_{i\in N^*}$ de la liste qui ……..
…….…….

* Supposons que $n_1 = 0$; la liste est …….., on n'entre pas dans la boucle et …….. est retourné (l'algorithme ……..)

* Si n_1>0 et vaut la longueur de la liste de départ : on entre dans la boucle, on compare v au 1er élément ;
  - soit il est égal à v et on ……. de la boucle en renvoyant ….. (et l'algorithme se termine) ;
  - soit il n'est pas égal à v et n_2 = ……

* A la k^{ième} itération, trois possibilités sont présentes :
  - soit v est égal au k^{ème} terme : on sort de la boucle en renvoyant ……..(et l'algorithme se termine)
  - soit v n'est pas égal au k^{ème} terme et $n_{k+1} = n_k - 1 = n_{k-1} - 2 = n_{k-2} - 3 = \ ... = n_1 - k + 1$
  - soit $n_k = 0$, tous les éléments de la liste ont été comparés à v sans succès, on sort de la boucle et …… est retourné (l'algorithme se termine donc)

* La suite strictement décroissante $(n_i)_{i\in\mathbb{N}^*}$ est minorée par …… donc d'après le cours de spécialité maths de terminale elle converge (on décrémente à chaque tour ce nombre d'une unité)

Complexité en temps : Dans le pire des cas (celui où v n'appartient pas à la liste), on parcourt l'ensemble de la liste de longueur n *:* on effectue n comparaisons, le temps de calcul de la fonction EST PROPORTIONNEL à la longueur de la liste. On parle d'une complexité en **O(n)** (lire « grand o de n ») : c'est un **coût linéaire**

Travail maison (à faire une fois la page précédente couverte en classe)

Soit un l'algorithme suivant :

v ← élément cherché

#valeur du nombre entier cherché
deb ← indice du premier élément de la liste
fin ← indice du dernier élément de la liste

r ← Faux #Réponse initialisée à Faux
i ← deb
Tant que i<= fin et que r=Faux faire
Si list[i] = v alors
r ← Vrai
FinSi
i ← i+1
FinTant que
Renvoyer la valeur de r

Analyser cet algorithme puis le faire tourner « à la main » sur la liste list=[19,2,81,70,7,97,85,26,45,86] comme dans l'exemple suivant en recherchant 2 d'abord puis 77 ensuite.

Exemple avec 81 _ v=81 r=Faux deb=0 et fin= 9 1 – list[0]= 19 ≠ v alors i=0+1 = 1 2- list[1]= 2 ≠v alors i=1+1 = 2 3- list[2]= 81 = v alors r=Vrai et i=2+1 = 3 4- on renvoie Vrai A vous v=2 r=Faux deb=0 et fin= 9 1 – list[0]= 19 ≠ v alors ……. 2- list[1] = 2 … v alors …….. 4- on renvoie …..	Avec 77 v=77 r=Faux deb=0 et fin= 9 1 – list[0]= 19 ≠ v alors i=0+1 = 1 2 – list[1]= 2 ≠ v alors i=1+1 = 2 …….. …….. …….. …….. …….. …….. …….. …….. …….. ……..

Complexité en temps

Combien de tours de boucles ont été nécessaires pour chercher 81 ? 2 ? et 77 ?

Afin d'estimer la complexité de cet algorithme, on se place dans le « pire des cas », c'est à dire le plus grand nombre de tours de boucle possible : décrivez les conditions de cette situation pour un nombre v donné et une liste donnée …….

Dans ce pire des cas, combien d'étapes seront nécessaires pour chercher v dans une liste de longueur n? …..

Fiche d'identité de l'algorithme de TRI par INSERTION

Principe : on parcourt les éléments de la liste ….....
……..→ tri en place

Exemple : le tri du jeu de cartes, on range une carte piochée dans son jeu déjà classé
https://www.youtube.com/watch?v=ejpFmtYM8Cw

Pseudo Code	**Python :**
n ← nombre d'éléments de t	.....
pour i allant de 1 à n-1	.....
tant que i>0 et t[i]<t[i-1]	.....
permuter t[i] et t[i-1]	.....
i ← i -1	.....
fin tant que	.....
fin pour	.....

Algorithme valide
On choisit comme invariant de boucle H : « la liste t[0 : i+1] est triée par ordre croissant à l'issue de l'itération i »
Initialisation (est-ce vrai avant d'entrer dans la boucle ?)
Conservation = hérédité (reste vraie après une itération, si elle était vraie avant)
Terminaison (donne le résultat attendu en fin de boucle)

Complexité en temps :
algorithme lent mais relativement efficace sur de courtes listes.
Dans le pire des cas, avec des données triées à l'envers, les parcours successifs du tableau imposent d'effectuer $(n-1)+(n-2)+(n-3)+\ldots+1=$ ……..(formule spé maths 1ère) comparaisons et échanges,
On a donc une complexité dans le pire des cas du tri par insertion en $O(n^2)$: c'est un coût quadratique

Exercice 1 – ANALYSER et CONCEVOIR
On souhaite trier la liste de listes Pers selon le numéro situé en 2ème position dans les sous-listes que contient Pers :

```
Pers =[['Portillon',4],['Sam',3],['Julie',1],['Tom',2],['Charlie',5]]
```

Proposer une fonction mettant en œuvre un algorithme de tri par insertion permettant de trier la liste Pers

Exercice 2 – CONCEVOIR et COMPARER
Afin de comparer les coûts en temps des méthodes de tri, proposer une fonction qui génère aléatoirement un tableau de taille n choisi par l'utilisateur d'entiers positifs.

Compléter à l'aide la fonction déjà utilisée lors de l'introduction du cours d'algorithmique

```
from time import time
debut = time()

# Code dont on mesure le temps

fin = time()
print("Temps passé : ",fin - debut)
```

Vu que le temps varie si on a un tableau initialement plus ou moins rangé, il sera intéressant de faire une moyenne sur plusieurs essais (en mettran la création des listes en dehors de la zone du programme servant à mesurer le temps écoulé.

Remplissez le tableau ce dessous et comparer avec vos camarades (selon la taille et la machine utilisée)

Taille n du tableau	Coefficients multiplicateurs	Temps Tri insertion	Coefficients multiplicateurs
10			
100			
1000			
5000			
10000			

Si la taille est multipliée par k, le temps de tri devrait théoriquement être multiplié par ….....
Est-ce que la réalité colle avec la théorie ? Pourquoi ?

Exercices

Exercice 1 – *CONCEVOIR ET TRADUIRE*

Écrire un algorithme qui compte le nombre d'éléments dans une liste, ce sans faire appel à une fonction dédiée (comme *len( )* en Python)

Indiquer la complexité temporelle si la liste contient n entiers.

Implémenter cet algorithme en Python

Exercice 2 – *CONCEVOIR ET TRADUIRE*

1°) Écrire un algorithme qui permette de trouver le minimum d'une liste d'entiers. Pour cela, on peut parcourir les éléments de la liste et utiliser mini, une variable initialisée au premier élément de la liste et n le nombre d'éléments de la liste

Estimer le coût en temps d'une telle recherche

Implémenter cet algorithme en Python

2°) Améliorer cet algorithme en *minimum_1indice(liste)* pour qu'il renvoie aussi le 1er indice du minimum rencontré

A FAIRE ABSOLUMENT **Créer un algorithme *minimum_indice(liste,j)* qui sera utile pour le tri séquentiel : il prend en paramètres la liste et un rang pour renvoyer l'indice du du minimum de la liste comprise entre les indices j et n-1**

3°) Créer enfin un algorithme *minimum_indiceS(liste)* qui renvoie TOUS les indices de chacune des occurrences du minimum dans la liste.

Estimer le coût en temps d'une telle recherche

Exercice 3 – *TRADUIRE ET DÉVELOPPER*

On associe des numéros à des personnages :

Personnage	Numéro
Céline	**1**
Lucas	**2**
Stéphane	**3**
Louison	**4**
Jean	**5**

Ceci se traduit en Python par la liste :

Pers = [['Celine',1],['Lucas',2],['Stephane',3],['Louison',4],['Jean',5]]

Écrire une fonction qui permet de rechercher si un personnage de numéro v (un entier) quelconque est présent dans la liste : la fonction renverra le nom du personnage alors et *False* sinon

Exercice 4 – *ANALYSER ET CONCEVOIR*

On prétend que la fonction suivante teste l'appartenance de la valeur v au tableau t

```
def appartient(v,t):
    for i in range(len(t)):
        if t[i] == v:
            trouvee = True
        else :
            trouvee = False
    return trouvee
```

Donner des tests pour cette fonction et en particulier des tests montrant plusieurs raisons pour laquelle cette fonction est incorrecte

Que teste cette fonction en réalité ?

Aller plus loin....et que pensez vous de

```
def appartient(v,t):
    for i in range(len(t)):
        if t[i] == v:
            trouvee = True
        else :
            trouvee = False
        return trouvee
```

(return a changé de place)

Fiche d'identité de l'algorithme de TRI par SÉLECTION (ou tri par extraction)

Principe : on va chercher le plus petit élément de la liste pour le mettre en …....., puis on repart du …..... élément et on va chercher le plus petit élément de la liste pour le mettre en …....., etc...

Exemple : le photographe, qui place les enfants sur le banc en fonction de leurs tailles, passe en revue les enfants à la recherche du plus petit. Une fois trouvé, il le place sur le banc. Il réitère cette opération sur les enfants restant en plaçant à chaque fois l'enfant sélectionné à la prochaine place disponible sur le banc.

Pseudo Code	**Python** :
n ← nombre d'éléments de t	….....
pour i allant de 1 à n-2	….....
i_min ← minimum(t,j)	…..... *// appel de la fonction minimum : renvoi son indice*
si i_min différent de j alors	….....
permuter t[i] et t[j]	….....
fin si	….....
fin pour	….....

Algorithme valide
On choisit comme invariant de boucle H : « la liste t[0 : i+1] est triée par ordre croissant à l'issue de l'itération i »
Initialisation (est ce vrai avant d'entrer dans la boucle ?)
Conservation = hérédité (reste vraie après une itération, si elle était vraie avant)
Terminaison (donne le résultat attendu en fin de boucle)

Complexité en temps : on s'intéresse au nombre de comparaisons effectuées. La liste est triée par ordre croissant. La boucle pour de la fonction tri effectue n-1 opérations. A chaque itération, la fonction minimum est appelée et effectue $n-(j+1)$ itérations. Sa complexité est donc en $O(n^2)$ de l'ordre de n×n

Formalisation
1°) Faire fonctionner à la main le tri par sélection sur un nouveau jeu de 4 cartes, puis sur une liste, [9,3,1,6] par exemple qui pourra être aussi illustrée si besoin par les 4 cartes de valeur correspondantes → Implémenter alors en pseudo code puis en Python cet algorithme
2°) Quel est le pire des cas ? Combien y-a-il alors d'itérations dans la boucle pour? Combien y-a-il d'appel à la fonction minimum dans chaque tour de boucle ? Combien d'itérations effectue cette dernière fonction ?
3°) Définir et trouver un Invariant de boucle

Exercices
Exercice 1 – ANALYSER et CONCEVOIR
On souhaite (ENCORE!) trier la liste de listes Pers selon le numéro situé en 2ème position dans les sous-listes que contient Pers :

```
Pers =[['Portillon',4],['Sam',3],['Julie',1],['Tom',2],['Charlie',5]]
```

Proposer une fonction mettant en œuvre un algorithme de tri par sélection permettant de trier la liste Pers

Exercice 3 – ANALYSER et TRADUIRE
En supposant que le tri par sélection prend un temps directement proportionnel à n2 et qu'il prend 6,8 secondes pour trier 16000 valeurs (voir tableau des temps de ce tri), calculer le temps qu'il faudrait pour trier un million de valeurs

Comparer ces temps avec les tris de Python

```
from time import time
debut = time()
# Code dont on mesure le temps
fin = time()
print("Temps passé : ",fin - debut)
```

On pourra générer aléatoirement des listes de 16000 éléments (liste_1) entre 1 et 10 000, puis de 1 million d'éléments (liste_2) toujours entre 1 et 10 0000
à l'aide du petit programme de comparaison d'écoulement du temps
comparer les temps de tri de liste.sort() ; un tri en place et sorted(liste) qui crée, elle, une copie triée dans Python.

Une nouvelle recherche....

Après la recherche séquentielle vue en début de chapitre, puis deux tris, nous allons de nouveau chercher un élément dans une liste mais cette fois-ci, TRIÉE ….

1°) Retrouvez le programme écrit en Python qui cache un nombre entier entre 1 et 100 et demande à l'utilisateur de le retrouver. https://www.dimension-k.com/maths/NSI/1NSI-(03-boucles)-jeu.py
On comptait même le nombre de coups (……..)
Observez et faire fonctionner le fichier Jeu dicho inversé depuis le site: décrire son rôle
https://www.dimension-k.com/maths/NSI/1NSI-(16-algorithmes-tri)-Jeu-dicho-inverse.py

2°) Nous allons nous intéresser toujours à une liste triée mais, désormais la liste n'est plus forcément une liste de nombres consécutifs.....

a) Comment repérer les éléments de cette liste et surtout comment repérer l'élément du milieu de la liste ?
….....

b) Faire fonctionner le même principe A LA MAIN afin de savoir si la valeur v = 81 est dans la liste triée suivante : list = [2 , 7 , 19 , 26 , 45 , 70 , 81 , 85 , 86 , 97]

Aide : m indice du milieu, deb l'indice du début de liste, fin celui de fin
Essayez de rédiger cela « à la main » : exemple ETAPE 1 – m=int((0+9)/2)=4 ,
list[4]=45 ≠ 81
comme list[m] < 81, alors deb = m+1
ETAPE 2 – m=int((5+9)/2)=7 , list[7]=85 ≠ 81 .....

c) Recommencez seul avec v= 77

Fiche d'identité de l'algorithme de RECHERCHE DICHOTOMIQUE

Principe : diviser pour régner
La méthode diviser pour régner consiste en trois étapes :
DIVISER : découper le problème initial en sous problèmes ;
RÉGNER : résoudre les sous problèmes
COMBINER : calculer une solution au problème initial à partir des solutions des sous problèmes
Dans un premier temps, la recherche par dichotomie va en fait trouver LA POSITION d'un élément dans la liste triée : il faut comparer l'élément avec la valeur de la case au milieu du tableau ; si les valeurs sont égales, la tâche est accomplie, sinon on recommence dans la moitié du tableau pertinente

Exemple : la recherche d'un mot dans le dictionnaire

Pseudo Code : à compléter	**Python** à implémenter
v ← élément cherché	……
deb ← indice du premier élément de la liste	……
fin ← indice du dernier élément de la liste	……
m ← ….....	……
Tant que deb <= fin faire	……
si v = …..... alors	……
renvoyer m	……
sinon si …..... alors	……
fin ← m–1	……
sinon	……
deb ← m+1	……
fin si	……
m ← ….....	……
Fin Tant que	……
Renvoyer Faux	……

TERMINAISON de l'algorithme :
Nous allons nous intéresser au fait que cet algorithme s'arrête…en effet, prouver ici que l'algorithme remplit bien son rôle (CORRECTION de l'algorithme) n'est pas trivial...

On cherche un variant de boucle dans une liste de n éléments

Il s'agit de démontrer que l'algorithme se termine dans tous les cas et qu'il n'y a pas de boucle infinie.
Dans ce cas précis, on peut choisir la suite $(fin_i - deb_i)_{i\in\mathbb{N}}$ des indices de fin et début de la liste considérée.

La condition d'arrêt de la boucle TANT QUE utilisée dans l'algorithme est Tant que deb<=fin mais on n'entre pas toujours dans la boucle :

Soit fin_i et deb_i avec i l'indice qui compte la ième itération de la boucle : deb_0 =..... et fin_0= …...
On note alors $m_i = arrondi\left(\frac{deb_i+fin_i}{2}\right)$
Si nous sommes à la k[ème] itération, il y a trois grandes possibilités :
- …..... ou ....., l'algorithme se termine, car la boucle n'est pas parcourue.
- $deb_k \ldots fin_k$ et $v < list[m_k]$, on "entre" dans la boucle : deb_{k+1}=..... et fin_{k+1} =..... : on a $fin_{k+1} - deb_{k+1} < fin_k - deb_k$
- deb_k fin_k et $v > list[m_k]$,on "entre" dans la boucle : deb_{k+1}=..... et fin_{k+1} =..... : on a $fin_{k+1} - deb_{k+1} < fin_k - deb_k$

On a donc chaque fois, $fin_{k+1} - deb_{k+1} < fin_k - deb_k$: la suite $(fin_i - deb_i)_{i\in\mathbb{N}}$ est strictement .....
Il existe donc un entier p de $\mathbb{N}$ tel que :
- soit $deb_p > fin_p$, l'algorithme va se …....., car on ne parcourt pas la boucle et l'algorithme renvoie ….....
- soit $v = list[m_p]$ avec $m_p = \frac{(deb_p+fin_p)}{2}$, l'algorithme va se terminer, car on parcourt la boucle et l'algorithme renvoie ….....

L'algorithme se termine donc toujours.

Complexité en temps : Le logarithme en base 2
DEFINITION : On le note $\log_2$ et pour x $\in\mathbb{R}$, $\log_2(2x) = x$

Lors d'une recherche dichotomique d'un élément v dans une liste triée « list » de longueur n, le nombre f de fois dans le cas le plus défavorable (v∉list) où la liste sera coupée en deux vérifie : $\frac{n}{2^f} = 1 \Leftrightarrow 2^f = n \Leftrightarrow f = \log_2(2^n)$

La complexité en temps dans le pire des cas de l'algorithme de recherche dichotomique est $O(\log_2(n))$ (ce qui est beaucoup mieux que un O(n²) comme la séquentielle).

Tracez les courbes $x \to \log_2(x)$ et $x \to x^2$ et comparer leurs croissances....

L'algorithme de recherche dichotomique est donc plus efficace que l'algorithme de recherche séquentielle car pour tout $x, x > \log_2(x)$. Néanmoins, il est à noter que la recherche dichotomique utilise un tableau TRIE, ce qui peut aussi avoir un coût en temps...

Exercices

Exercice 1 – ANALYSER
Combien de valeurs sont examinées lors d'un appel à la dichotomie sur la liste [0, 1, 2, 2, 3, 6, 9, 13, 13, 20] pour trouver 7 ? Et avec un appel à une recherche séquentielle ?

Exercice 2 – TRADUIRE ET DÉVELOPPER
Écrire en Python une fonction Repetition (n) qui calcule et renvoie le plus petit entier f tel que $2^f > n$.

Rappeler ce que représente f dans une recherche dichotomique dans une liste de taille n.

residus
tri selection
Exercice 2 – ANALYSER et TRADUIRE
On souhaite maintenant trier la liste de listes Pers selon le numéro situé en 2ème position dans les sous-listes que contient Pers :

```
Pers =[['Sam',3],['Julie',1],['Tom',2],['Portillon',4],['Charlie',5]]
```

Mais cette fois ci, seul un élément est mal trié. Il s'agit de la liste ['Sam',3] qui correspond à l'indice 0 dans la liste Pers . Proposer une fonction mettant en œuvre un algorithme de tri conçu par vous même permettant de trier la liste Pers.
On ne remplacera que l'élément mal placé, sans trier toute la liste.
(cette situation peut être illustrée par le rangement d'un livre selon son titre dans une bibliothèque dont les ouvrages sont classés par ordre alphabétique : on ne trie pas toute la bibliothèque mais on insère l'ouvrage à sa place)

Aides : del Pers[0] supprime l'élément mal placé
Pers.insert(i,mal_placee) insère la liste mal_placee (à déterminer) à l'indice i (à déterminer)