2I003 : Ordonnancement d'une application sur une machine hétérogène

Serge DURAND et Arthur LIMBOUR

January 20, 2019

Toutes les fonctions décrites dans ce rapport sont fournis dans un fichier zip en annexe, contenant les codes sources en python.

1 Exercice 1 : Implémentation d'un premier algorithme

1.1 Question 1

1.1.1 1 et 2 : calculs "à la main"

Méthode : on calcule d'abord les valeurs pour les feuilles, puis pour un parent on prend le maximum de (la valeur des feuilles + l'éventuel temps de communication) et on y ajoute le temps d'exécution du parent. En effet toutes les feuilles partageant le même parent sont exécutées en même temps, donc ce qui détermine la durée d'exécution est le temps maximum parmi les feuilles + le temps de communication.

u	0	1	2	3	4	5	6	7
$\Delta(u)$	13	7	1	7	2	3	5	7
s(u)	0	2	8	6	7	5	8	8

1.1.2 3 : Preuve de $\mathcal{P}$

On reprend les notations de l'énoncé.

Base :

Soit $u \in V$ une tâche isolée, donc une arborescence réduite à un sommet, $u \in ABT(V)$. On a donc $\mathcal{G}_u = (u)$, c'est à dire $\Gamma^+(u) = \emptyset$ et pour une allocation donnée, la durée d'exécution minimale est simplement la durée d'exécution $t_{\sigma(u)}(u)$, donc $\mathcal{P}(u)$ est vérifiée.

Induction :

Soient $u \in V$ et $\Gamma^+(u) = \{v_1, ..., v_p\} \neq \emptyset$.
On suppose que pour tout v de $\Gamma^+(u)$ on a $\mathcal{P}(v)$ vraie (**hypothèse d'induction**).
Soit $\mathcal{G}_{v_1}, \mathcal{G}_{v_2}, ..., \mathcal{G}_{v_p}$ la suite d'arborescences de $ABT(V)$ telle que $\mathcal{G}_{v_i}$ est l'arborescence de durée minimale $\Delta(v_i)$ pour tout $i \in \{1, ..., p\}$.
Alors $\forall i \in \{1, ..., p\}, u \notin \mathcal{G}_{v_i}$, donc $\mathcal{G}_u = (u, \mathcal{G}_{v_1}, ..., \mathcal{G}_{v_p}) \in ABT(V)$ (par définition inductive de $ABT(V)$).
Toutes les tâches $v \in \Gamma^+(u)$ peuvent être exécutée en simultanément et donc la durée minimale d'exécution de $\mathcal{G}_u = \Delta(u)$ vaut : $max_{v \in \Gamma^+(u)}\{\Delta(v) + c_{\sigma(u)\sigma(v)}((u, v))\} + t_{\sigma(u)}(u)$, car une fois la tâche u effectuée il faut exécuter les tâches de chaque arborescence et l'exécution totale s'achève pour après l'exécution des tâches de l'arborescence $\mathcal{G}_{v_i}$ avec v_i tel que $\Delta(v_i) + c_{\sigma(u)\sigma(v_i)}((u, v_i)) = max_{v \in \Gamma^+(u)}\{\Delta(v) + c_{\sigma(u)\sigma(v)}((u, v))\}$, donc $\mathcal{P}(u)$ est vérifiée.

Conclusion :

On a montré que $\mathcal{P}(u)$ vraie pour toute tâche isolée u puis que si $\mathcal{P}(v)$ vraie pour tout $v \in \Gamma^+(u)$ alors $\mathcal{P}(u)$ est vraie, donc par induction structurelle sur $ABT(V)$, la propriété est vraie pour tout $u \in V$.

1.2 Question 2

Pour les complexités des différentes fonctions, n est le nombre de tâches de l'arborescence étudiée, c'est aussi la longueur du tableau Arbre.

1.2.1 Tableaux de données

```
Arbre = [0,0,3,0,1,1,3,3]
tA = [2,3,3,4,10,3,20,5]
tB = [3,10,1,2,2,15,5,1]
cAB = [0,1,2,4,2,3,3,3]
cBA = [0,4,3,3,4,2,3,4]
```

Note : pour les tableaux cAB et cBA on a choisit de mettre la valeur 0 pour le sommet 0, puisqu'il n'y a jamais de temps de communication pour la tâche effectuée en premier. Il n'y a aucune arête $(u, 0)$ dans l'arbre.

1.2.2 fonction duree(u,Sigmau,tA,tB)

```
def duree(u,Sigmau,tA,tB):
    if(Sigmau==1):
        return tA[u]
    return tB[u]
```

Complexité en $\Theta(1)$: on fait une seule comparaison, et l'accès à un tableau est en $\Theta(1)$.

1.2.3 fonction valCom(u,v, Sigmau,Sigmav,cAB,cBA)

```
def valCom(u,v,Sigmau,Sigmav,cAB,cBA):
    if(Sigmau==Sigmav):
        return 0
    elif(Sigmau==1):
        return cAB[v]
    return cBA[v]
```

Complexité en $\Theta(1)$ également, au plus 2 comparaisons, accès à cAB ou cBA en temps constant.

1.3 Question 3 : fonction succ(u,Arbre)

```
def succ(u,Arbre):
    res = set()
    for i in range(1,len(Arbre)):
        if(Arbre[i] == u):
            res.add(i)
    return res
```

Complexité (en comptant le nombre de comparaison) en $\Theta(n)$: on fait exactement n-1 comparaisons (il n'est pas nécessaire de de comparer Arbre[0] qui n'est le successeur de personne). La primitive "add" est appelée au plus (n-1) fois (cas où u = 0 et l'arbre est de hauteur 2). La complexité de "add" est en $\Theta(1)$.
Remarque : si on avait choisit de compter le nombre d'appels à add la complexité serait en $\mathcal{O}(n)$, mais l'instruction élémentaire de comparaison est plus significative car effectuée plus de fois que la primitive add. La primitive add étant en temps constant (comme la comparaison) son appel ne change pas la classe de complexité. On a choisit le set plutôt que la liste pour l'utilisation dans l'exercice 2. Par définition de nos arbres les successeurs d'un seul sommet sont uniques, le set ne pose pas de problème.

1.4 Question 4 : CalculDelta

On calcule la durée minimale d'exécution de $\mathcal{G}_u$ récursivement, sur le mode de la formule de la question 1 : le cas de base : u est une tâche isolée (une feuille de l'arbre), dans ce cas la fonction renvoie $t_{\sigma(u)}(u)$. Sinon on appelle la fonction sur les successeurs de u. On s'est servi de la syntaxe des compréhensions de liste pour parcourir les successeurs.

```
def CalculDelta(Arbre,tA,tB,cAB,cBA,sigma,u):
    Succ = succ(u,Arbre)
    if(len(Succ)==0):
        return duree(u,sigma[u],tA,tB)
    return duree(u,sigma[u],tA,tB) + max([CalculDelta(Arbre,tA,tB,cAB,cBA,sigma,v)+valCom(u,v,
    sigma[u],sigma[v],cAB,cBA) for v in Succ])
```

Complexité : On fait au plus n appels : dans le cas où la fonction est appelée pour $u = 0$, sur la racine de l'arbre, il faut parcourir toutes les tâches de l'arborescence. A chaque appel récursif, (et même au premier appel) on appelle une fois la fonction succ, en $\Theta(n)$, puis si l'on est pas dans le cas de base, on fait exactement 2 additions, un appel à la primitive "max" et $k = Card(\Gamma^+(u))$ appels à valCom : on l'appelle sur chaque successeur de u. De plus la complexité de la primitive max est en $\Theta(k)$ puisque la taille du tableau sur laquelle elle est appelée est k. Finalement la complexité peut être calculée par le nombre d'appels à succ car $k \leq n$ dans tout les cas.
Dans le pire cas il y a n appels à succ, qui est en $\Theta(n)$. La complexité de la fonction est donc quadratique : au plus $n * \Theta(n)$ donc $\mathcal{O}(n^2)$.

1.5 Question 5 : CalculSigmaOptimum

Fonctions auxiliaires :
(on travaille sur les bits plutôt qu'avec les opérations arithmétiques pour tenter d'optimiser un peu... un left shift de 1 bit est par exemple une division par 2).

```
def convBinaire(n,l):
    res =[]
    while(n!=0):
        res.append(n&1)
        n = n>>1
    res.reverse()
    #on remplit par des 0 à gauche le reste de la liste
    res = [0]*(l-len(res)) + res
    return res

def CalculListeAllocs(Arbre):
    n = len(Arbre)
    res = []
    for i in range(2**n-1):
        res.append(convBinaire(i,n))
    return res
```

Fonction principale :

```
def CalculSigmaOptimum(Arbre,tA,tB,cAB,cBA):
    Allocs = CalculListeAllocs(Arbre)
    #Allocs = [list(tup) for tup in list(itertools.product(range(2), repeat=n))]
    Lbest = Allocs[0]
    Deltabest = CalculDelta(Arbre,tA,tB,cAB,cBA,Allocs[0],0) #vu qu'on recherche un min on ne peut
     pas initialiser à 0
    for sigma in Allocs[1:]: #pas besoin de recalculer pour la première allocation possible
        Deltatemp = CalculDelta(Arbre,tA,tB,cAB,cBA,sigma,0)
        if Deltatemp < Deltabest: #on a trouvé une meilleure allocation
            Deltabest = Deltatemp
            Lbest= sigma
    return (Lbest,Deltabest)
```

Complexité :

Il y a 2^n allocations possibles. On peut le voir en remarquant que l'ensemble des allocations est en bijection avec l'ensemble des fonctions de $\{1, ..., n\}$ dans $\{0, 1\}$, et ce dernier ensemble a pour cardinal : $Card(\{0,1\})^{Card(\{1,...,n\})} = 2^n$. Il y a alors 2^n appels à CalculDelta, la complexité est donc $\Theta(n^2 * 2^n)$. Ici on est bien en Θ car on sait que la fonction CalculDelta est appelée d'abord sur la racine à chaque allocation. Note sur la constitution des allocations : nous avons testé la bibliothèque itertools pour alléger le code, mais le problème de mémoire est alors rencontré pour la même valeur de h, il n'y a pas d'améliorations notables de performance.

2 Exercice 2 : Second Algorithme

2.1 Question 1

Méthode : on calcule d'abord les deux valeurs pour chaque feuille puis on remonte l'arbre jusqu'à la racine.

u	0	1	2	3	4	5	6	7
$D^{\mathcal{A}}(u)$	13	7	3	12	10	3	20	5
$D^{\mathcal{B}}(u)$	14	15	1	7	2	15	5	1

2.2 Question 2

On note $\mathcal{P}_2(u)$ la propriété : $(D^{\mathcal{A}}(u) \leq \Delta^{\mathcal{A}}(u)$ et $D^{\mathcal{B}}(u) \leq \Delta^{\mathcal{B}}(u))$.

Base :

Soit $u \in V$ une tâche isolée, donc $\mathcal{G}_u = (u)$ et $\Gamma^+(u) = \emptyset$. On a alors $D^{\mathcal{A}}(u) = t_{\mathcal{A}}(u)$ et $D^{\mathcal{B}}(u) = t_{\mathcal{B}}(u)$.

On a aussi $\Delta^{\mathcal{A}}(u) = t_{\mathcal{A}}(u)$ et $\Delta^{\mathcal{B}}(u) = t_{\mathcal{B}}(u)$ puisque le seul ordonnancement réalisable possible est la simple exécution de la tâche u. Alors $D^{\mathcal{A}}(u) \leq \Delta^{\mathcal{A}}(u)$ et $D^{\mathcal{B}}(u) \leq \Delta^{\mathcal{B}}(u)$

Induction :

Soient $u \in V$ et $\Gamma^+(u) = \{v_1, ..., v_p\} \neq \emptyset$. On suppose que pour tout v de $\Gamma^+(u)$ on a $\mathcal{P}_2(v)$ vraie (**hypothèse d'induction**). Donc, $\forall i \in \{1, ..., p\}$, $D^{\mathcal{A}}(v_i) \leq \Delta^{\mathcal{A}}(v_i)$ et $D^{\mathcal{B}}(v_i) \leq \Delta^{\mathcal{B}}(v_i)$.
Soit $\mathcal{G}_{v_1}, \mathcal{G}_{v_2}, ..., \mathcal{G}_{v_p}$ la suite d'arborescences de $ABT(V)$ telle que $\mathcal{G}_{v_i}$ est l'arborescence de durée minimale $\Delta(v_i)$ pour tout $i \in \{1, ..., p\}$.
On note $\mathcal{G}_u^{\mathcal{A}}$ l'ordonnancement tel que $\sigma(u) = \mathcal{A}$ et $\forall v_i \in \Gamma+(u)$, $\mathcal{G}_{v_i}$ est l'ordonnancement optimal de l'arborescence avec v_i en sommet. On a donc $\Delta^{\mathcal{A}}(u)$ la durée d'exécution de l'ordonnancement $\mathcal{G}_u^{\mathcal{A}}$. D'autre part $(u, \mathcal{G}_{v_1}, ..., \mathcal{G}_{v_p}) \in ABT(V)$ par définition inductive de $ABT(V)$.
Soit $k \in \{1, ..., p\}$ tel que $\min(D^{\mathcal{A}}(v_k), c_{\mathcal{AB}}((u, v_k)) + D^{\mathcal{B}}(v_k)) = \max_{v_i \in \Gamma^+(u)} \min(D^{\mathcal{A}}(v_i), c_{\mathcal{AB}}((u, v_i)) + D^{\mathcal{B}}(v_i))$.
Autrement dit, v_k est le fils de u pour lequel l'exécution de l'ordonnancement des tâches + l'éventuel temps de communication entre u et v_k est la durée la plus longue parmi tout les fils de u (c'est lui qui déterminera donc la durée d'exécution de $\mathcal{G}_u^{\mathcal{A}}$, puisque tout les v_i peuvent être exécutée simultanément).
On suppose d'abord que $\min(D^{\mathcal{A}}(v_k), c_{\mathcal{AB}}((u, v_k)) + D^{\mathcal{B}}(v_k)) = D^{\mathcal{A}}(v_k)$. On a donc $\sigma(v_k) = \mathcal{A}$ et :
$D^{\mathcal{A}}(u) = t_{\mathcal{A}}(u) + D^{\mathcal{A}}(v_k) \Rightarrow D^{\mathcal{A}}(u) \leq t_{\mathcal{A}}(u) + \Delta^{\mathcal{A}}(v_k)$ par hypothèse d'induction.
D'où $D^{\mathcal{A}}(u) \leq \Delta^{\mathcal{A}}(u)$.

Sinon on a $\min(D^{\mathcal{A}}(v_k), c_{\mathcal{AB}}((u, v_k)) + D^{\mathcal{B}}(v_k)) = c_{\mathcal{AB}}((u, v_k)) + D^{\mathcal{B}}(v_k)$, donc $\sigma(v_k) = \mathcal{B}$. Alors on a :
$D^{\mathcal{A}}(u) = t_{\mathcal{A}}(u) + c_{\mathcal{AB}}((u, v_k)) + D^{\mathcal{B}}(v_k) \Rightarrow D^{\mathcal{A}}(u) \leq t_{\mathcal{A}}(u) + c_{\mathcal{AB}}((u, v_k)) + \Delta^{\mathcal{B}}(v_k)$ par hypothèse d'induction, donc $D^{\mathcal{A}}(u) \leq \Delta^{\mathcal{A}}(u)$ dans les deux cas possible.

On montre de manière analogue, en considérant $\mathcal{G}_u^{\mathcal{B}}$, que dans tout les cas $D^{\mathcal{B}}(u) \leq \Delta^{\mathcal{B}}(u)$. Donc $\mathcal{P}_2(u)$ est vérifiée.

Conclusion :

On a montré que $\mathcal{P}_2(u)$ vraie pour toute tâche isolée u puis que si $\mathcal{P}(v)$ vraie pour tout$v \in \Gamma^+(u)$ alors $\mathcal{P}(u)$ est vraie, donc par induction structurelle sur $ABT(V)$, la propriété est vraie pour tout $u \in V$.

2.3 Question 3

2.3.1 Principe

On commence par les feuilles et on remonte jusqu'à la racine. On fait attention à chaque étape de ne calculer des valeurs $D^{\mathcal{A}}$ et $D^{\mathcal{B}}$ uniquement pour les tâches dont ces deux valeurs sont connues **pour tout les successeurs**. Pour cela on ajoute en paramètre de la fonction un ensemble qui contient les sommets explorés (**Marques**), mis à jour à chaque appel, et un ensemble de candidats (**Candidats**) qui contient les sommets dont on veut calculer les valeurs $D^{\mathcal{A}}$ et $D^{\mathcal{B}}$. Fonctionnement :

1. Au premier appel on calcule les feuilles puis les valeurs $D^{\mathcal{A}}$ et $D^{\mathcal{B}}$ de chaque feuille. On insère les feuilles dans l'ensemble des candidats explorés (Marques), qui est vide pour le premier appel. On détecte que c'est le premier appel si la liste des candidats est de même taille que l'arbre. Par construction les candidats ne peuvent être l'arbre entier pour tout les appels suivants. On utilise la souplesse de python sur le typage : pour le premier appel Candidats est une liste mais pour les suivants c'est un ensemble. Les syntaxes utilisées (for u in L, ou len(Candidats)) étant applicables aux deux types. On construit un ensemble de candidats pour le nouvel appel (Ncandidats) en y insérant les parents de chaque feuille si leurs successeurs sont tous dans l'ensemble Marques. En effet si ce n'est pas le cas il manquera des valeurs pour pouvoir calculer $D^{\mathcal{A}}$ et $D^{\mathcal{B}}$.
2. Lors des appels suivant, si l'on est pas encore dans le dernier appel, on calcule d'abord $D^{\mathcal{A}}$ et $D^{\mathcal{B}}$ pour nos candidats, puis on les ajoute à l'ensemble des candidats explorés et on construit les nouveaux candidats comme dans le premier appel. L'ensemble des candidats explorés est conservé entre chaque appel mais celui des nouveaux candidats est mis à 0 au début de chaque appel. On sait qu'on est dans un appel intermédiaire si Candidats n'est ni l'ensemble réduit à la tâche 0, ni tout les noeuds de l'arbre.
3. Au dernier appel il ne reste plus qu'à calculer $D^{\mathcal{A}}$ et $D^{\mathcal{B}}$ pour 0, et faire un retour pour pouvoir sortir de la fonction.

2.3.2 Code (version commentée en annexe)

Note : le compteur cpt n'est là que pour calculer le nombre d'appels.

```
def CalculBorneinf(Arbre,Candidats,Marques,tA,tB,cAB,cBA,dA,dB,cpt):
    """Au premier appel Candidats = Arbre, Marques = set vide.
    dA = dB = [0]*len(Arbre)
    cpt n'est là que pour compter les appels récursifs, on le met à 0...
    """
    if(Candidats=={0} or Candidats==[0]):
        dA[0]=max([min([dA[v],dB[v]+cAB[v]]) for v in succ(0,Arbre)]) + tA[0]
        dB[0] = max([min([dB[v],dA[v]+cBA[v]]) for v in succ(0,Arbre)]) + tB[0]
        Marques.add(0).
        cpt += 1
        print(cpt)
        return (dA,dB)

    NCandidats = set()
    if(len(Arbre)==len(Candidats)):
        Candidats = Feuille(Arbre)
        for u in Candidats:
            dA[u] = tA[u]
            dB[u] = tB[u]
            if(succ(Arbre[u],Arbre)<=Candidats):
                NCandidats.add(Arbre[u])
        cpt += 1
        CalculBorneinf(Arbre,NCandidats,Candidats,tA,tB,cAB,cBA,dA,dB,cpt)
    else:
        for u in Candidats:
            dA[u] = max([min([dA[v],dB[v]+cAB[v]]) for v in succ(u,Arbre)]) + tA[u]
            dB[u] = max([min([dB[v],dA[v]+cBA[v]]) for v in succ(u,Arbre)]) + tB[u]
            Marques.add(u)
            if(succ(Arbre[u],Arbre)<=Marques):
                NCandidats.add(Arbre[u])
        cpt += 1
        CalculBorneinf(Arbre,NCandidats,Marques,tA,tB,cAB,cBA,dA,dB,cpt)
```

Fonction auxiliaire pour le calcul des feuilles :

```python
def Feuille(Arbre):
    L = [0]*len(Arbre)
    res= set()
    for i in range(len(Arbre)):
        L[Arbre[i]] = 1
    for i in range(len(Arbre)):
        if(L[i]==0):
            res.add(i)

    return res
```

2.3.3 Complexité

1. La complexité de la fonction Feuille(Arbre) est linéaire : on parcourt exactement deux fois l'arbre, et on ne fait que deux opérations maximum à chaque tour de boucle (une affectation dans une liste pour la première, une comparaison et éventuellement un ajout dans un ensemble pour la deuxième). La fonction Feuille n'est appelée qu'une fois dans la fonction CalculBorneInf.

2. On fait exactement h appels récursifs, avec h la hauteur de l'arbre. En effet on parcourt l'arbre niveau par niveau en partant des feuilles. L'arbre n'a pas de structure particulière dans les hypothèses du sujet, donc dans le pire cas on fait n appels récursif (arbre en ligne droite, chaque sommet a un seul fils).

3. Soit $k_i = Card(Candidats)$ pour le i-ème appel récursif. La primitive add et le test d'inclusion $S1 \leq S2$ (c'est la syntaxe en python) pour deux sets sont respectivement en $\Theta(1)$ et $\mathcal{O}(len(S1))$.
   Ici aussi les k_i forment une partition de n (en considérant l'ensemble des feuilles et non l'arbre pour le premier appel), pour n'importe quelle structure d'arbre (par exemple si h = 2, $k_1 = n-1$ et $k_2 = 1$).
   Pour le i-ème appel on fait $3*k_i$ appels à la fonction succ (on pourrait en faire deux en déclarant une variable Succ = succ(u)...), car on appelle 3 fois la fonction sur chaque candidat. La fonction succ étant linéaire c'est elle qui importe pour la complexité finale.

Au final la complexité est : $(\sum_{i=1}^{h} *k_i * 3 * \Theta(n)) = (\sum_{i=1}^{h} k_i) * 3 * \Theta(n) = 3n * \Theta(n) = \Theta(n^2)$

2.4 Question 4

Ici il n'y a pas de difficulté particulière. On créé une fonction auxiliaire qui calcule l'allocation d'un sommet donné en paramètre. Elle distingue le cas où le sommet est 0. Puis la fonction principale appelle cette fonction auxiliaire pour tout les sommets de l'arbre grâce à un parcours en profondeur, écrit récursivement.

2.4.1 code

```python
def CalculSigmaOptimumAux(Arbre,tA,tB,cAB,cBA,dA,dB,Sigma,v):
    if(v==0):
        if (dA[0] <= dB[0]):
            Sigma[0]=1
        return Sigma[v]
    if(Sigma[Arbre[v]]==1):#u est dans A
        if(dA[v]<(dB[v]+cAB[v])):
            Sigma[v]=1
        else:
            Sigma[v]=0
        return Sigma[v]
    else:#u est dans B
        if(dB[v]>=(dA[v]+cBA[v])): #on inverse le test par rapport au sujet
            Sigma[v]=1
        else:
            Sigma[v]=0
        return Sigma[v]

```

```
19  def CalculSigmaOptimum2(Arbre,tA,tB,cAB,cBA,dA,dB,Sigma,Marques,u):
20      """on considère sigma initialisé à 0 partout
21      Marques est ici une liste de booleen : Marques[u] = 0 indique que Sigma[u] n'a pas été calculé
        """
22
23      Marques[u]=1
24      Sigma[u] = CalculSigmaOptimumAux(Arbre,tA,tB,cAB,cBA,dA,dB,Sigma,u)
25      for v in succ(u,Arbre):
26          if(Marques[v]==0):
27              CalculSigmaOptimum2(Arbre,tA,tB,cAB,cBA,dA,dB,Sigma,Marques,v)
```

Il suffit ensuite d'appeler cette dernière fonction pour u = 0.

2.4.2 Complexité

La fonction auxiliaire est en temps constant : il n'y a aucune boucle, aucune fonction auxiliaire, il n'y a qu'un nombre fini, indépendant des paramètres, d'instruction de base.
La fonction CalculSigmaOptimum2 fait exactement n appels récursifs grâce au suivi des sommets explorés, et à chaque appel elle invoque une fois la fonction auxiliaire et une fois la fonction succ. La complexité est donc quadratique. Au final pour obtenir une allocation optimale on appelle une fois CalculBorneinf et une fois CalculSigmaOptimum2 : **la complexité totale est quadratique.**

3 Exercice 3 : mesure expérimentale de complexité

3.1 Question 1

Tout est en annexe. Pour le jeu de tests l'allocation trouvée n'est pas nécessairement la même pour chaque fonction, mais dans tout les cas elle est optimale, la durée d'exécution minimale est la même. Il est en effet possible d'avoir plusieurs allocations ayant une même durée d'exécution.

3.2 Question 2

Le code est laissé en annexe. On utilise la bibliothèque random et on a créé une troisième fonction qui renvoie une instance complète avec un arbre et ses données. On a écrit des fonctions pour créé des arbres droits et des arbres de hauteur 2 également. Les arbres droits servent à affiner les tests pour l'algorithme 1.

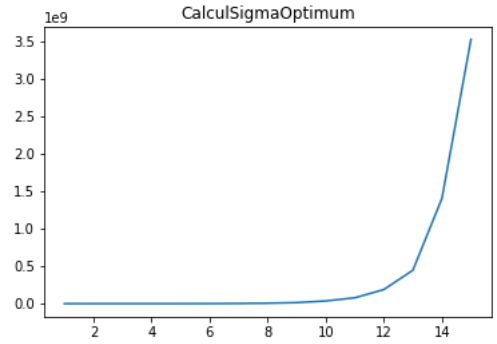
3.3 Question 3

Pour CalculSigmaOptimum on rencontre un problème de mémoire pour $h = 5$, c'est à dire $n = 31$. Le nombre d'opérations étant alors de l'ordre de $2^{31} = 2147483648$ opérations. En faisant des tests intermédiaires sur des arbres droits on affine : le problème de mémoire est rencontré dès n = 19.
On écrit une fonction qui fait le calcul d'allocation par l'algorithme 2 directement (elle appelle simplement une fois CalculBorneinf puis CalculSigmaOptimum2 en ayant initialisé des listes à 0 partout pour dA, dB etc.). Pour cette fonction la dernière valeur de h pour laquelle l'algorithme fonctionne est h =15, donc n = 32767. A ce stade l'exécution est déjà de 2-3 minutes, il ne nous semble pas nécessaire d'affiner plus. Pour les mesures expérimentales on prendra h = 12 au plus car on répétera les tests pour une même valeur de n plusieurs fois afin de lisser les valeurs.

3.4 Question 4

Pour cette fonction on ne peut tester que jusqu'à h = 4 si on teste sur des arbres complets. Pour obtenir plus de points intermédiaires on teste sur des arbres droits jusqu'à n = 15 plutôt que sur 4 points comme suggéré par l'énoncé.

Voilà le graphe obtenu :

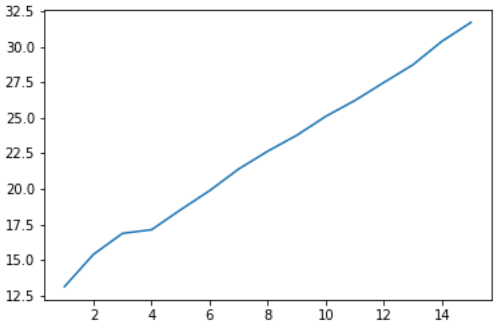


On reconnaît la courbe d'une fonction exponentielle. Les valeurs sont ($T_1(n)$ est en ns) :

n	1	2	3	4	5	6	7	8	9	10
$T_1(n)$	8960	43306	120532	143786	379732	967678	2773116	6613111	14353475	36163793

n	11	12	13	14	15
$T_1(n)$	78240966	188379492	446402842	1404290682	3526867806

En prenant le log en base 2 des $T_1(n)$ on obtient le graphe suivant :

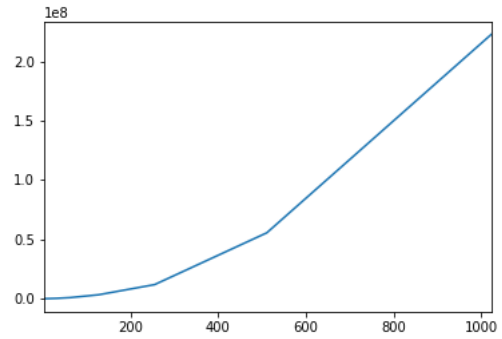


La courbe est bien linéaire. On calcule sa pente en calculant la pente entre chaque points consécutifs des données obtenues et en prenant la médiane de toutes les pentes, elle vaut environ 1.33.

```
def CalculPente(Lx,Ly):
    """Lx contient les abscisses (ici les tailles des arbres testés)
    Ly les ordonnées (ici le temps d'exécution)"""
    res = []
    for i in range(len(Lx)-2):
        pente = (Ly[i+1]-Ly[i])/(Lx[i+1]-Lx[i])
        res.append(pente)
    pente = np.median(res)
    return pente
```

3.5 Question 5

Voilà le graphe obtenu pour le test sur des hauteurs de 2 à 12, donc des tailles de 3 à 4095 :

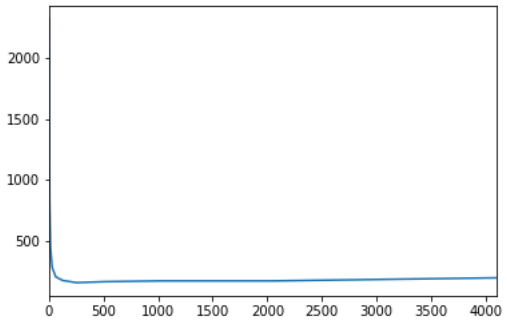


La courbe semble bien quadratique.

n	3	7	15	31	63	127	255	511	1023	2047	4095
$T_2(n)$	2	4.1	10	27.2	81.9	283	1026	4335	18011	72130	331045

Dans le tableau les valeurs de $T_2(n)$ ont été divisée par 10000 pour un tableau plus présentable.

Voilà la courbe si on divise chaque $T_2(n)$ par n^2 :



Pour des grandes valeurs de n la fonction semble bien constante, ce qui confirme que la complexité est quadratique.