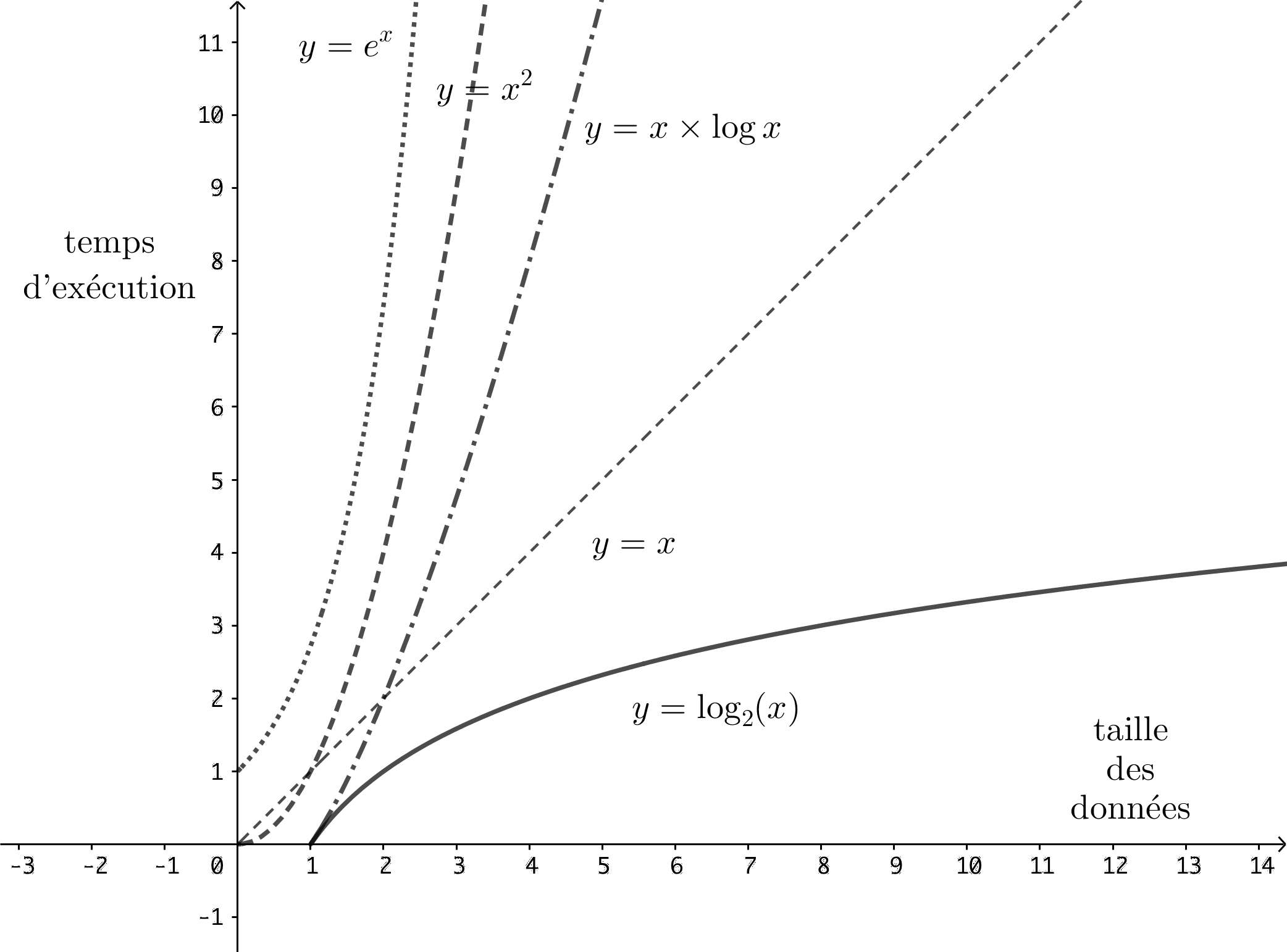
Diviser pour Regner

Principe :

* Diviser : partager le problème en sous-problèmes de même nature (souvent de taille *n* /2)
* Régner : résoudre ces différents sous-problèmes (généralement récursivement)
* Combiner : fusionner les solutions pour obtenir la solution du problème initial

Avantage : gain en complexité

*Rappel pour les calculs de complexité* : on peut voir le logarithme entier en base deux d’un entier comme le nombre de chiffres nécessaire à son écriture en base 2, ou bien, de manière équivalente, comme l’exposant de la plus petite puissance de 2 strictement supérieure.

Par exemple,  et 

Exemple simple : juste prix en récursif

*Exercice introductif*: l’utilisateur donne un nombre à deviner à l’ordinateur. Programmer la fonction récursive justePrix (borne\_inf, borne\_sup, prix), qui permet à l’ordinateur de trouver le plus rapidement possible le nombre à trouver.

Signature de la fonction

def justePrix(borne\_inf, borne\_sup, prix) :

"""

Trouve récursivement un nombre donné par l’utilisateur

@param borne\_inf, borne\_sup, prix : entiers vérifiant borne\_inf ≤ prix ≤ borne\_sup

"""

La complexité de l’algorithme que vous avez codé est, si vous avez codé un algorithme efficace, . En effet, la taille de l’intervalle [borne\_inf...borne\_sup] est divisée par deux à chaque appel de la fonction récursive.

Si on déroule le fonctionnement de l’algorithme à l’envers, on multiplie la taille de l’intervalle par deux à chaque étape :

Fin : prix = borne inf = born sup de 1ongueur 1

[borne\_inf...borne\_sup] de longueur 2

[borne\_inf...borne\_sup] de longueur 4

…

[borne\_inf...borne\_sup] de longueur  borne\_sup - borne\_inf

On retrouve bien la définition du logarithme entier en base 2 telle que donnée ci-dessus.

Exemple : le tri fusion

Rappel : les tris connus, par sélection sélection ou par insertion, sont de complexité quadratique,  . Ce paragraphe va donner un exemple de tri plus efficace.

*Principe*:

* on se donne une liste de longueur *n*, d’objets de même type, comparables.
* on scinde la liste en deux sous-listes de même longueur (à 1 près), de manière récursive
* jusqu’à obtenir des listes de longueur 1. Qui sont triées : avec un seul élément, ce n’est pas trop dur…
* On remonte la récursivité en fusionnant deux listes « voisines » de manière à conserver l’ordre sur le résultat.

*Exemple*:

* Fonctionnement général de l’algorithme

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  | 8 | 7 | -3 | 0 | 4 | 12 | 2 |  |  |  |
|  |  |  |  |  |  | ↙︎ |  |  | ↘ | ︎ |  |  |  |
| Division 1 |  |  | 8 | 7 | -3 | 0 |  |  | 4 | 12 | 2 |  |  |
|  |  |  |  | ↙︎ | ↘ |  |  |  | ↓ |  | ↘ |  |  |
| Division 2 |  | 8 | 7 |  |  | -3 | 0 |  | 4 | 12 |  | 2 |  |
|  |  | ↙︎ | ↓ |  |  | ↙︎ | ↓ |  | ↓ | ↘ |  |  | ↘ |
| Division 3 | 8 |  | 7 |  | -3 |  | 0 |  | 4 |  | 12 |  | 2 |
|  |  | ↘ | ↓ |  |  | ↘ | ↓ |  | ↓ | ↙︎ |  |  | ↙︎ |
| Fusion 1 |  | 7 | 8 |  |  | -3 | 0 |  | 4 | 12 |  | 2 |  |
|  |  |  |  | ↘ |  | ↓ |  |  | ↘ |  | ↙︎ |  |  |
| Fusion 2 |  |  | -3 | 0 | 7 | 8 |  |  | 2 | 4 | 12 |  |  |
|  |  |  |  |  | ↘ |  |  |  | ↙︎ | ︎ |  |  |  |
| Fusion 3 |  |  |  | -3 | 0 | 2 | 4 | 7 | 8 | 12 |  |  |  |

* Détails de la phase de combinaison.

On parcourt les deux sous-listes de gauche et de droite en même temps, en triant les éléments au fur et à mesure.

Supposons que le programme en soit au moment où il doit combiner les tableaux [7, 8] et [-3, 0].

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  | Résultat | | | |
| Début | 7 | 8 |  | -3 | 0 |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |
| Comparaison 1 : -3 < 7 | 7 | 8 |  | -3 | 0 |  | -3 |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |
| Comparaison 2 : 0 < 7 | 7 | 8 |  | -3 | 0 |  | -3 | 0 |  |  |
|  |  |  |  |  |  |  |  |  |  |  |
| La liste de droite a été entièrement explorée, | 7 | 8 |  | -3 | 0 |  | -3 | 0 | 7 | 8 |
| on recopie les éléments restants à gauche |  |  |  |  |  |  |  |  |  |  |

*Complexité*:

Comme on divise la taille du tableau par 2 à chaque appel récursif, on fait  appels. Lors de la phase de combinaison, on parcourt les listes de gauche et de droite, donc en temps *n* à chaque appel récursif. D’où une complexité quasi linéaire .

*Remarque pour la programmation* : utiliser des slices [a : b] n’est pas obligatoire, on peut très bien recopier les listes élément par élément, notamment avec des listes par compréhension.

*Programme :*

def tri\_fusion(liste) :

"""

Sépare la liste en deux

Et appelle récursivement sur les sous-listes la fusion/combinaison des résultats

"""

if len(liste) <= 1 :

return liste

else :

milieu = len(liste)//2

gauche = liste[:milieu]

droite = liste[milieu:]

# ou

# gauche = [liste[i] for i in range(milieu)]

# droite = [liste[i] for i in range(milieu, len(liste)]

return fusion(tri\_fusion(gauche), tri\_fusion(droite))

def fusion(gauche, droite) :

"""

Fusionne deux listes triées en seule liste triée

"""

resultat = []

indice\_gauche = 0

indice\_droite = 0

while indice\_gauche < len(gauche) and indice\_droite < len(droite):

if gauche[indice\_gauche] < droite[indice\_droite]:

resultat.append(gauche[indice\_gauche])

indice\_gauche = indice\_gauche +1

else :

resultat.append(droite[indice\_droite])

indice\_droite = indice\_droite +1

if indice\_gauche == len(gauche):

while indice\_droite != len(droite):

resultat.append(droite[indice\_droite])

indice\_droite = indice\_droite + 1

if indice\_droite == len(droite):

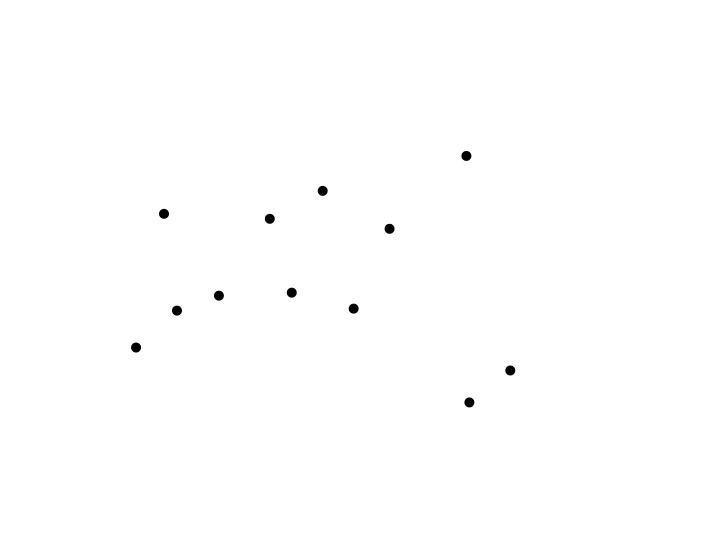
while indice\_gauche != len(gauche):

resultat.append(gauche[indice\_gauche])

indice\_gauche = indice\_gauche + 1

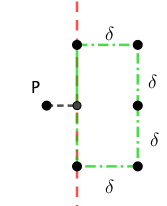
return resultat

Exemple visuel : recherche des deux points les plus proches dans un nuage planaire

On dispose de points dans un plan. On cherche quels sont les deux points les plus proches. Par force brute, on ordonne les points par coordonnée *x* croissante (tri fusion ou tri rapide de complexité  ). Puis, pour chaque point  d’abscisse , on calcule toutes les distances  avec *j* ≥ *i*. On retrouve un « classique » du calcul de complexité, à savoir que le nombre d’itérations est  , soit une complexité quadratique  (on rappelle que dans ce cas le coût du tri en  est négligeable en comparaison).

Avec diviser pour régner : on commence par créer deux tableaux ordonnés avec les points. L’un, , est ordonné suivant les abscisses *x* croissantes, l’autre  est ordonné suivant les ordonnées *y* croissantes. On a donc une complexité 

|  |  |
| --- | --- |
| MacintoshHD:Users:frederic:Fred:T NSI:cours:tnsi_07_divers:tnsi_07_dr_dist_points_2.png | MacintoshHD:Users:frederic:Fred:T NSI:cours:tnsi_07_divers:tnsi_07_dr_dist_points_3.png |
| Descente récursive 1 (et suivantes):  Avec  , on scinde l’ensemble de points en deux parties de taille égale à 1 près. | Descente récursive 2 et suivantes :  On scinde chaque sous-ensemble en deux parties de taille égale à 1 près (toujours avec ) |
| MacintoshHD:Users:frederic:Fred:T NSI:cours:tnsi_07_divers:tnsi_07_dr_dist_points_4.png | MacintoshHD:Users:frederic:Fred:T NSI:cours:tnsi_07_divers:tnsi_07_dr_dist_points_5.png |
| Fin de la descente récursive :  Dans chaque ensemble de 2 ou 3 points, on résout le problème directement. Avec 2 points il n’y a qu’une seule possibilité, avec 3 points on fait 2 comparaisons. On obtient  dans chaque partie. | Phase de combinaison :  On crée une bande de largeur  , où . Avec ** de chaque côté de la bande médiane |
| MacintoshHD:Users:frederic:Fred:T NSI:cours:tnsi_07_divers:tnsi_07_dr_dist_points_6.png | MacintoshHD:Users:frederic:Fred:T NSI:cours:tnsi_07_divers:tnsi_07_dr_dist_points_7.png |
| Dans cette bande, pour chaque point à gauche, on calcule les distances avec points de droites qui sont « autour » de ce point dans le tableau  (cf ① ci-dessous). On prend le minimum de ces distances | La phase de combinaison se termine en prenant le minimum des trois distances  ,  , |
| MacintoshHD:Users:frederic:Fred:T NSI:cours:tnsi_07_divers:tnsi_07_dr_dist_points_8.png | MacintoshHD:Users:frederic:Fred:T NSI:cours:tnsi_07_divers:tnsi_07_dr_dist_points_9.png |
| On itère le processus en remontant la récursivité. Ici on a représenté le calcul de certaines distances à cheval sur droite et gauche | Fin. |

① Lors de ce calcul, pour un point donné de gauche d’ordonnée  , on se contente de calculer les distances avec les points de droite situés dans le rectangle centré en ordonnée  , de hauteur (ordonnées) 2*d*, et de largeur *d*. On a au maximum 8 points dans ce rectangle (aux coins et  au milieu de chaque grand côté, en considérant deux carrés côte à côte). Un de ces points du milieu étant alors P. Donc on a maximum 7 points concernés.

On trouve ces points par une recherche dichotomique dans le tableau trié  (complexité ), on cherche le premier point dans  : tel que  , et n’étant pas à gauche. Puis on calcule les distances sur les 7 points suivants de droite dans , ou bien on calcule les distances tant que .

*Complexité*: Lors de la phase de combinaison, la recherche dans le tableau  par dichotomie est de complexité , et on répète 6 fois les calculs de distance au plus, pour chaque point de gauche. Donc la complexité est  . Ajoutant à cela la division par 2 à chaque étape récursive, la complexité de cet algorithme est  (y compris les tris initiaux).

Exercices

Exercice 1 : Recherche du couple (minimum, maximum) dans un tableau avec une méthode « diviser pour régner ».

On scindera le tableau en deux parties et on effectuera la recherche récursivement.

Si le tableau a une taille de 2, alors le couple (minimum, maximum) s’obtient directement par comparaison des deux valeurs.

Si le tableau a une taille de 1, alors le couple (minimum, maximum) est composé de deux fois l’unique valeur du tableau.

Lors de la phase de combinaison, on comparera les résultats « remontant » de la récursivité pour obtenir le résultat.

Écrire le programme qui donne ce couple. Donner la complexité de l’algorithme.

*Remarque* : ceux qui sont à l’aise essaieront autant que possible de ne pas utiliser de slices, mais plutôt des indices de début et de fin. Ceci pour des raisons d’efficacité lors de l’implémentation.

Exercice 2 : Recherche du sous-tableau de somme maximale dans un tableau contentant des nombres positifs et négatifs, avec une méthode « diviser pour régner ».

On donne un tableau contenant des nombres positifs et négatifs. On veut trouver la somme maximale d’éléments consécutifs de ce tableau

*Exemple* : le tableau est T = [-2, -5, **6, -2, -3, 1, 5**, -6]. La somme maximale est 7, correspondant aux nombres en gras.

Principe de l’algorithme :

Si le tableau T a 1 élément :

Résoudre directement le problème (pas très compliqué normalement…)

Sinon :

Diviser T en deux sous-tableaux T1 et T2

Résoudre le problème sur chacun des tableaux T1 et T2

Renvoyer la meilleure des trois solutions (sur T1, sur T2, et à cheval sur T1 et T2). Pour la solution à cheval, on part du milieu (la fin de T1 / respectivement le début de T2), et on cherche la somme maximale à gauche (respectivement à droite), puis on ajoute les deux.

Faire tourner cet algorithme « à la main » sur l’exemple donné, puis le programmer. On pourra créer une fonction chargée de calculer la somme maximale à cheval sur les deux tableaux. Il n’est pas indispensable d’écrire une fonction spécifique pour la combinaison des résultats.

Donner la complexité de l’algorithme, et comparer avec le calcul par force brute, où l’on calcule toutes les distance possibles.

*Remarques/compléments* :

* Cet exercice permet de trouver le couple (T[i] , T[j]) dans un tableau, tel que T[j] – T[i] soit maximal (avec i < j). Comment, c’est une bonne question.
* *Remarquez* la similitude avec l’exemple de la recherche du couple de points les plus proches dans un plan, à deux dimensions. Cet exercice est dans une seule dimension.
* l’algorithme de recherche d’un couple de points de distance minimale dans un nuage se généralise en dimension quelconque. Sans chercher à écrire le programme, expliquer la méthode en dimension 3 est un excellent exercice.

Exercice 3 *(source WIkipedia)* :Algorithme de Karatsuba (mathématicien soviétique, 1937 – 2008)

Cet algorithme permet d’effectuer plus rapidement une multiplication que la méthode usuelle, notamment dans le cas de grands nombres.

Le calcul naïf d’un produit est :  . Vérifiez sur un exemple, comme , c’est ce que vous faites lorsque vous posez la multiplication. Ce calcul demande quatre produits : *ac*, *ad*, *bc* et *bd*. Quand les deux nombres font *n* chiffres, la complexité est quadratique :.

En 1960, Karatsuba remarque que le calcul peut être fait avec seulement trois produits :



**Vérifiez** que ça fonctionne. Au moins de tête !

La méthode est appliquée de manière récursive pour les calculs de *ac*, *bd*, et  en scindant à nouveau *a,* *b*, *c* et *d* en deux.

Comme la multiplication par 10 pour les humains et par 2 pour les machines correspond à l’ajout d’un 0 en fin, et que les additions sont peu coûteuses en temps, le gain de complexité n’est pas négligeable. On obtient une complexité de  . Pour des nombres de 1000 chiffres, on passe ainsi de 1000000 de multiplications avec la méthode traditionelle à seulement 50000 avec l’algorithme de Karatsuba. Le coût des additions est négligeable.

*Exemple*:

* Descente récursivité étape 1



* Descente récursivité étape 2 + remontée étape 2



* Remontée récursivité étape 1 :



* Le calcul a demandé 9 produits au lieu de 16 par la méthode traditionnelle. Bien sûr, à la main c’est fastidieux. Mais pas pour une machine.

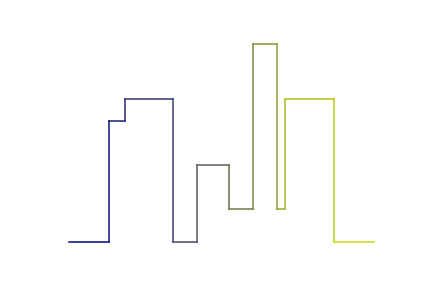
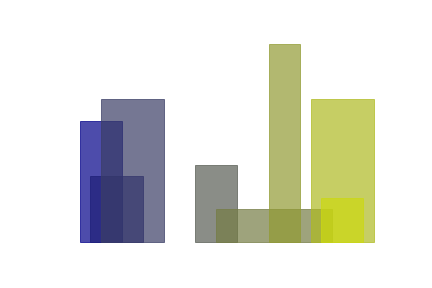
Programmer la fonction karatsuba(x, y, n), où x et y sont deux entiers strictement positifs de n chiffres. Attention pour l’appel récursif : on n’utilise pas n//2 ici, mais ceil(n/2) (la valeur supérieure et non inférieure), à importer de la bibliothèque math. Pourquoi ? réfléchissez.

Exercice 4 : dessiner la ligne d’horizon dans une ville.

Dans une ville vue en deux dimensions, les immeubles sont donnés par un triplet de coordonnées (gauche, hauteur, droite). On veut dessiner la ligne d’horizon, c’est-à-dire ne montrer que les lignes qui sont visibles, et cacher celles qui sont derrière des immeubles.

Le programme va renvoyer une liste de segments horizon. Ces segments sont donnés par un couple (gauche, hauteur). On peut remarquer qu’un tel segment a comme limite droite la coordonnée gauche de son successeur (sauf pour celui le plus à droite, on verra ultérieurement comment préciser cette limite)

On va construire cette ligne d’horizon par une méthode « diviser pour régner ».



*Remarque* : comme dans les exercices précédents, cet exercice peut se faire par une méthode directe. La complexité est en  contre  avec diviser pour régner.

*Données*: les immeubles seront donnés par ordre de coordonnée gauche croissante.

*Exemple :* exemple visualisé ci-dessus

immeubles = [(1, 11, 5), (2, 6, 7), (3, 13, 9), (12, 7, 16), (14, 3, 25),

(19, 18, 22), (23, 13, 29), (24, 4, 28)]

*Exemple pour la phase de fusion*(elle ressemble à celle du tri-fusion) :

On a deux lignes d’horizon h1 et h2. Ce sont les lignes d’horizon de la dernière phase de fusion correspondant à la liste d’immeubles ci-dessus.

h1 = [(1, 11), (3, 13), (9, 0), (12, 7), (16, 0)]

h2 = [(14, 3), (19, 18), (22, 3), (23, 13), (29, 0)]

Et une ligne d’horizon résultante, initialement vide

hr = []

Deux variables hauteur1 et hauteur2 sont initialisées à 0. hauteur1 sera mise à jour quand un segment de h1 est ajouté à hr, et hauteur2 sera mise à jour quand un segment de h2 est ajouté à hr.

* On compare (1, 11) et (14, 3). Le premier segment a pour coordonnée x = 1 < 14, on l’ajoute dans le résultat. On augmente l’indice de parcours de h1 de 1.

hr = [(1, 11)] hauteur1 = 11

* On fait de même avec (3, 13), (9, 0) et (12, 7) (comparés avec (14, 3))

hr = [(1, 11) (3, 13), (9, 0), (12, 7)] hauteur1 = 7

* On fait de même avec (16, 0) et (14, 3). Ici, c’est le deuxième segment, de h2, qui a la coordonnée x la plus petite : 16 > 14

On ajoute donc un segment de coordonnées x = 14 dans hr. Comme on ajoute un segment de h2, on met à jour hauteur2, et on calcule la hauteur du segment à rajouter en prenant le maximum des deux hauteurs.

hauteur2 = 3

nouvelle hauteur = max (hauteur1, hauteur2) = max (7, 3) = 7

D’où hr = [(1, 11) (3, 13), (9, 0), (12, 7), (14, 7)]

* On reprend à nouveau (16, 0)et on le compare avec (19, 18). Comme 16 < 19, on ajoute le segment à partir du x de h1. La variable hauteur1 est mise à jour : hauteur1 = 0

Donc nouvelle hauteur = max (hauteur1, hauteur2) = max (0, 3) = 3

Puis hr = [(1, 11) (3, 13), (9, 0), (12, 7), (14, 7), (16,3)]

* h1 a été entièrement parcouru, on ajoute les segments restants de h2

hr = [(1, 11) (3, 13), (9, 0), (12, 7), (14, 7), (16,3),

(19, 18), (22, 3), (23, 13), (29, 0)]

* On remarque que deux segments consécutifs, (12, 7) et (14, 7), ont la même hauteur. On nettoie hr en enlevant le deuxième.

hr = [(1, 11) (3, 13), (9, 0), (12, 7), (16,3),

(19, 18), (22, 3), (23, 13), (29, 0)]

* Dans le code, plutôt que de nettoyer la liste a posteriori, il est plus simple (et moins coûteux en temps suivant l’implémentation) de ne pas rajouter un segment si le segment précédent est de la même hauteur

*Un fichier python d’utilitaires est donné (utilitaires\_ligne\_horizone.py). Il doit être dans le même dossier que le fichier « tnsi\_07\_dr\_horizon\_exo.py », qui est à compléter. Ce dernier vous permet d’avoir les spécifications des fonctions ainsi qu’un affichage du résultat*

1. Programmer la fonction de fusion comme ci-dessus. On l’appelle fusionSkyLine(h1, h2). Ne pas oublier le cas où on finit de parcourir h2 avant h1. Ce qui en théorie, vu la manière dont on fait les appels récursifs, ne devrait pas se produire. Ceci dit : « en théorie, la théorie et la pratique c’est pareil, mais en pratique, c’est pas vrai ».
2. Programmer la fonction récursive horizonRec(immeubles) qui va construire la ligne d’horizon par appel de fusionSkyLine(h1, h2). Réfléchissez bien à ce qu’on doit renvoyer quand il y a un seul immeuble, c’est là qu’est la subtilité.
3. Pour ceux qui ont du temps.
   1. Créer une fonction qui génère aléatoirement une liste d’immeubles correctement triée, et avec un aspect esthétique si possible.
   2. Sans regarder la solution (qui est dans le fichier d’utilitaires), construire la fonction qui, étant donné la liste hr, renvoie les coordonnées des points qui permettent le tracé de la ligne. Comparer la complexité spatiale entre hr et cette liste de points.
   3. Pour ceux qui ont vraiment beaucoup de temps, faire la version avec des toits pointus/obliques (l’idée paraît sympathique, votre prof vous fait confiance pour trouver). A priori, la seule difficulté, pas énorme, est de trouver les points d’intersection. Il faut également changer la structure des triplets immeubles en (gauche, hauteur gauche, hauteur milieu, hauteur droite, droite), ce qui couvre tous les cas possibles. On rentre ici dans le domaine de l’art génératif ([https://fr.wikipedia.org/wiki/Art\_génératif](https://fr.wikipedia.org/wiki/Art_g%C3%A9n%C3%A9ratif)). C’est un sujet éventuel pour le grand oral.

Cours de Frédéric Mandon et Clément Michaut, licence Creative Commons BY NC SA, <https://creativecommons.org/licenses/by-nc-sa/3.0/fr/>

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |
|  |  |