ACT

Compte Rendu

TP1

Pierre Lague & Romain Lecleire

Abstract

Ce premier TP aborde un problème d'algorithmique simple. L'objectif est de familiariser les étudiants à réfléchir à différentes solutions tout en considérant la question de la complexité temporelle. La paradigme "Diviser pour régner" sera étudié dans ce TP. Le langage de programmation utilisé est `Python 3`.

Q.1 Une première approche brute-force

En s'aidant du sujet, nous pouvons exprimerla surface maximale du rectangle en respectant les contraintes de la façon suivante : $aireMax = (x_j - x_j) * y_{min}$. Avec y_{min}, la plus petite ordonnée par rapport à h (la hauteur totale) permettant de constuire un rectangle d'aire maximale ne contenant aucun points. Nous pouvons en déduire un algorithme qui consiste à parcourir les points par paire afin de constuire tous les rectangles possibles.

Section 1: Algorithme en $O(n^3)$

L'algorithme suivant décrit ce processus :

Algorithm 1 Calcul de l'aire maximale du rectangle ne contenant aucun des points en $O(n^3)$

```
1:  max_area ← 0
2:  for i ← 0 to len(points) − 1 do
3:      xt1 ← points[i][0]
4:      yt1 ← points[i][1]
5:      for j ← i + 1 to len(points) − 1 do
6:          xtj ← points[j][0]
7:          ytj ← points[j][1]
8:          min_y ← h                          ▷ h : hauteur donnée dans le fichier test
9:          for k ← i + 1 to j − 1 do
10:             xtk ← points[k][0]
11:             ytk ← points[k][1]
12:             if min_y > ytk then            ▷ vérification de l'ordonnée la plus petite formant un "candidat"
13:                 min_y ← ytk                ▷ mise à jour du minimum
14:             end if
15:         end for
16:         res ← (xtj − xt1) · min_y          ▷ calcul de l'aire du "candidat "
17:         max_area ← max(res, max_area) ▷ Si cette aire est plus grande que la précédente, mise à jour de
    l'aire maximale
18:     end for
19: end for
```

L'analyse des boucles de l'algorithme nous permet de déduire les points suivants :

- La boucle en i itère de 0 à $len(points) - 1$. Elle à n itérations avec n le nombre de points. Elle est en $O(n)$.
- La boucle en j itère de $i + 1$ à $len(points) - 1$. Elle à également n itérations. Elle est en $O(n)$.
- La boucle en k itère de $i + 1$ à $j - 1$. En entrant dans cette boucle, i et j sont fixes, donc au pire des cas elle à n itérations. Elle est en $O(n)$.

Section 2: Analyse de la complexité $O(n^3)$

En utilisant les notations de sommes on obtient :

$$
\begin{aligned}
S &= \sum_{i=0}^{n-1} \sum_{j=i+1}^{n-1} \sum_{k=i+1}^{j-1} \\
&= \sum_{i=1}^{n} \sum_{j=i}^{n} \sum_{k=i}^{j} \\
S &= \sum_{i=1}^{n} \sum_{j=i}^{n} (j-i+1) \quad \text{changement de variable } k = j-i+1 \\
&= \sum_{i=1}^{n} \sum_{k=1}^{n-i+1} k \\
&= \sum_{i=1}^{n} \frac{(n-i+1)(n-i+2)}{2} \quad \text{changement de variable } m = n-i+1 \\
&= \sum_{m=1}^{n} \frac{m(m+1)}{2} \\
&\dots \\
&= \Theta\left(n^3\right)
\end{aligned}
$$

En faisant tourner l'algorithme sur les données de test, on retrouve les bons résultats dans un temps d'éxécution relativement rapide (pour N0, N2, N10 et les deux N100). Cepedant, en essayant N500, l'exécution est très longue, preuve que notre algorithme n'est pas "efficace" dans le sens de la compléxité temporelle.

Section 3: Algorithme en $O(n^2)$

Il nous est donc demandé d'implémenter un algorithme en $O(n^2)$ afin d'avoir des temps d'éxécution plus courts. Il est possible de simplifier l'algorithme en $O(n^3)$ en calculant au préalable la valeur de y_{min} ce qui nous évite de faire une boucle k.

L'algorithme suivant décrit ce processus.

Algorithm 2 Calcul de l'aire maximale du rectangle ne contenant aucun des points en $O(n^2)$

```
1:  max_area ← 0
2:  for i ← 0 to len(points) − 1 do
3:      xt1 ← points[i][0]
4:      yt1 ← points[i][1]
5:      min_y ← h
6:      for j ← i + 1 to len(points) − 1 do
7:          xtj ← points[j][0]
8:          ytj ← points[j][1]
9:          if ytj < min_y and j ≠ i + 1 and i − j < 0 then
10:             min_y ← ytj
11:         end if
12:         res ← (xtj − xt1) · min_y
13:         max_area ← max(res, max_area)
14:     end for
15: end for
```

L'analyse des boucles de l'algorithme nous permet de déduire les points suivants :

- La première boucle en i à n itérations.

- La seconde boucle en j itère de $i+1$ à $n-1$. Donc à au pire n itérations.

Section 4: Analyse de la compléxité $O(n^2)$

En utilisant les notations de sommes on obtient :

$$\begin{aligned} S &= \sum_{i=0}^{n-1} \sum_{j=i+1}^{n-1} \\ &= \sum_{i=1}^{n} \sum_{j=i}^{n} \\ &= \sum_{i=1}^{n} n-i+1 \quad \text{avec} \quad k = n-i+1 \\ &= \sum_{k=1}^{n} k = \frac{n(n+1)}{2} \\ &... \\ &= \Theta\left(n^2\right) \end{aligned}$$

Une approche avec le paradigme "Diviser pour Régner"

Afin d'améliorer davantage notre solution, il est possible d'implémenter un programme en utilisant le paradigme algorithmique 'Diviser Pour Régner' qui consiste à diviser le problème initial en plus petits problèmes, les résoudre et reconstituer la solution de base avec les solutions des sous-problèmes.

L'algorithme suivant utilise ce concept :

Algorithm 3 Calcul de l'aire maximale du rectangle ne contenant aucun des points en $O(n)$

```
function DiviserPourReigner(h, g, d)
    min_y ← h
    min_index ← −1
    if d = g + 1 then
        res ← (points[d][0] − points[g][0]) · min_y
        return res
    end if
    for i ← g + 1 to d − 1 do
        if min_y > points[i][1] then
            min_y ← points[i][1]
            min_index ← i
        end if
    end for
    res ← max(RectangleDivide(h, g, min_index), RectangleDivide(h, min_index, d), (points[d][0] −
points[g][0]) · min_y)
    return res
end function
```

Section 5: Analyse de la complexité $O(n^2)$

Appels récursifs : L'algorithme effectue deux appels récursifs :

- $rectangle\_div2(h, g, min\_index)$
- $rectangle\_div2(h, min\_index, d)$

Chacun de ces appels récursifs opère sur une plage plus petite de g et d. Dans le pire des cas, l'algorithme peut effectuer ces appels récursifs pour chaque segment du rectangle, créant ainsi un arbre binaire d'appels récursifs.

Boucle : L'algorithme contient une boucle qui itère de g + 1 à d - 1. Le nombre d'itérations dans cette boucle dépend de la différence entre d et g. Dans le pire des cas n.

L'algorithme à une structure de type "diviser pour régner", similaire à une recherche binaire. À chaque niveau de récursivité, la taille du problème est réduite d'un facteur 2 (en supposant que min_index divise le rectangle à peu près en deux). Sachant que la recherche préalable du minimum coûte n opérations, on s'approche d'une compléxité en $O(n^2)$

Dans le meilleur des cas, l'algorithme vérifie si d est égal à g + 1 dès le début. Si cette condition est remplie, il calcule le résultat et le renvoie sans effectuer d'autres appels récursifs ou itérations. En somme, la complexité temporelle dans le meilleur des cas est O(1) lorsque l'algorithme peut rapidement identifier le cas de base et renvoyer le résultat sans effectuer d'appels récursifs ou d'itérations de boucle supplémentaires. Toutefois, ce scénario optimal suppose une disposition spécifique des points qui conduit à une évaluation rapide du cas de base. Dans la pratique, les performances réelles dépendront des données d'entrée et de leur distribution.

Comparaison de performances

En calculant le temps d'éxécution des différents algorithmes, nous avons généré ce tableau renseignant les performances des différentes solutions au problème.

Taille/Complexité	O(n^3)	O(n^2)	D2R O(n^2))	Multi-Threading
10	1.2*10^-4	5.86*10^-5	6.05*10^-5	4.9*10^-3
100	0.20	3.2*10^-3	1.0*10^-3	0.031
500	1.76	0.053	1.6*10^-3	0.16
100000	NA	NA	Recursive Limit Error	143.6

On observe qu'en utilisant le paradigme "Diviser pour Regner", on est en $O(n^2)$ mais on a des temps d'éxécution qui sont plus rapides que l'algorithme en $O(n^2)$ sans le paradigme. Ceci est du à l'utilisation de la récursivité.

Section 6: Diviser pour Regner en Parallèle

Listing 1: Python Code Example

```
# Le code ci-dessous est a titre experimental pour les threads. Nous ne nous sommes
 ↪ pas vraiment penches sur les questions du sujet.
#
# CODE COPIE POUR RENDRE LES THREAD PLUS SIMPLE A INTEGRER
#
class NewThread(Thread):
    def __init__(self, group=None, target=None, name=None,
                 args=(), kwargs={}):
        Thread.__init__(self, group, target, name, args, kwargs)
    def run(self):
        if self._target != None:
            self._return = self._target(*self._args, **self._kwargs)
    def join(self, *args):
        Thread.join(self, *args)
        return self._return

def rectangle_div_thread(h,g,d):

    min_y = h
    min_index = -1

    if d == g + 1:
        return (points[d][0] - points[g][0]) * min_y

    for i in range(g+1,d):
        if min_y > points[i][1] :
            min_y = points[i][1]
            min_index = i

    res1 = NewThread(target=rectangle_div_thread,args=(h,g,min_index))
    res2 = NewThread(target=rectangle_div_thread,args=(h,min_index,d))
    res1.start()
    res2.start()

    res3 = (points[d][0] - points[g][0]) * min_y
    return max(res1.join(),res2.join(),res3)
```