Pseudocode de l'algorithme et analyse de la complexité

La complexité de la fonction Run dépend uniquement que de celles de MutliTowersVorace et Tabou.

```
1  def Run(blocks):
2      towers = MutliTowersVorace(blocks)
3      towers = Tabou(towers)
4      if towers.size < bestTowers.size:
5          bestTowers = towers
```

MutliTowersVorace contient une boucle appelant la fonction Vorace un certains nombres de fois jusqu'à ce que la liste de bloques soit vide.

```
1  def MutliTowersVorace(blocks):
2      while !blocks.empty():
3          towers.append(Vorace(blocks))
4      return towers
```

La fonction Vorace quant à elle fait un tri à la ligne 2. On peut assumer que ce tri soit d'une complexité de $nlog(n)$ puisque cela correspond à la complexité de std::sort. Une boulce for est ensuite utilisé ce qui représente une complexité linéaire. De ce fait, nous estimons que la complexité en pire cas de la combination de la boucle de MultiTowersVorace et celle est de Vorace est n^2.

```
1  def Vorace(blocks)
2      blocks.sort(criterion)
3      tower.append(blocks.front())
4      blocks.removeFirst()
5
6      for block in blocks:
7          if tower.height + block.height >= maxHeight:
8              break
9          if block.fitOn(tower.last()):
10             tower.append(block)
11             blocks.remove(block)
12
13     return tower
```

La fonction tabou commence par trouver la plus petite tour. Cela implique simplement d'itérer au travers des tours donc la complexité est linéaire au nombre de tour. Par la suite, tabou fait appel à FindBestTowerToInsertInto qui représente la plus grande complexité de la fonction. Les opérations suivantes sont en pire cas de complexité linéaire.

```
def Tabou(towers):
    smallestTower = FindSmallestTower(towers)
    blockToMove = smallestTower.pop()

    blocksToRemove, tower, position = FindBestTowerToInsertInto(
        towers, blockToMove, smallestTower)

    if smallesTower.empty():
     towers.remove(smallestTower)

    tower.insert(position, blockToMove)
    tower.updateTabous()
    tower.tabou(blocksToRemove)
    tower.remove(blocksToRemove)

    newTower = createTower(blocksToRemove)
    towers.append(newTower)
```

La fonction FindBestTowerToInserInto effectue une boucle itérant sur les tours puis une autre itérant sur les blocs de chaque tour pour trouver la position d'insertion. Par la suite, elle itére sur les blocs se situant au-dessus de la position d'insertion pour trouver lesquels ne peuvent pas tenir sur le bloc à insérer. Toutes ces opérations représentent, en pire cas, une complexité du nombre de tour multiplié par le nombre de bloc par tour. Pour simplifier, nous pouvons simplement dire qu'ils s'agit d'une complexité n^2. Par la suite, la fonction fait appel à trimTower.

```
def FindBestTowerToInsertInto(towers, blockToMove, smallestTower):
    for tower in towers:
        if tower == smallest tower:
            continue

        if tower.isBlockTabou(blockToMove):
            continue

        positionToInsertAt = tower.findWhereBlockFit(blocksToMove)
        for block in blocksAfterPositionToInsertAt:
            if !block.fitOn(blockToMove):
                blocksToRemove.append(block)
            else:
                break

        blocksToRemove = TrimTower(tower, blockToMove,
            blocksToRemove)

        if blocksToRemove.size <= smallestBlocksToRemove.size:
            isBetterSolution = true

            if blocksToRemove.size == smallestBlocksToRemove.size:
                squareHeight =
                    (tower.height - blocksToRemove.height +
                        blockToMove.height)^2
                    + blocksToRemove.height^2
                    + (smallestTower.height - blockToMove.height)^2
                if squareHeight > bestSolutionSquareHeight:
                    bestSolutionSquareHeight = squareHeight
```

```
28                  else:
29                      isBetteSolution = false
30
31              if isBetterSolution:
32                  smallestBlocksToRemove = blocksToRemove
33
34      return smallestBlocksToRemove
```

TrimTower contient une boucle itérant jusqu'à ce que la hauteur de la tour soit plus petite ou égale au maximum permis. Cette boucle fait appel à la fonction popTallestBlock qui trouve le plus grand bloc dans la tour. Cette fonction effectue son travail avec une complexité linéaire. En pire cas, la fonction aura donc une complexité de n^2. Cependant, le pire cas ne pourrait arriver que si le bloc ajouté est aussi ou plus grand que la somme des hauteurs des blocs déjà dans la tour.

```
1  def TrimTower(tower, blockToMove, blocksToRemove)
2      while tower.height + blockToMove.height - blocksToRemove.height
           > maxHeight:
3          blocksToRemove.append(tower.popTallestBlock())
4
5      return blocksToRemove
```

En conclusion, en pire cas, la fonction tabou aurait une complexité égale au nombre de tour multiplié par n^2. Cependant, ce cas devrait être assez rare. Alors, nous considérons qu'en moyenne la complexité de l'algorithme est probablement plus proche de n^2.