—Schrebergärten—

TEAM-ID: 01020
GRUPPE: PLEPPOS
RAYKO EICHHORST, THEO JACOBI UND KIRU SPREU
23.11.2018

Anfänglich ist es essentiell eine Agenda zu haben, nach der man strukturiert agieren kann. Prinzipiell ist das System stets gleich.
Die ersten paar Schritte sind gewissermaßen in den theoretischen Teil einzugliedern, worauf danach die praktische Implementierung und Exekution folgt.

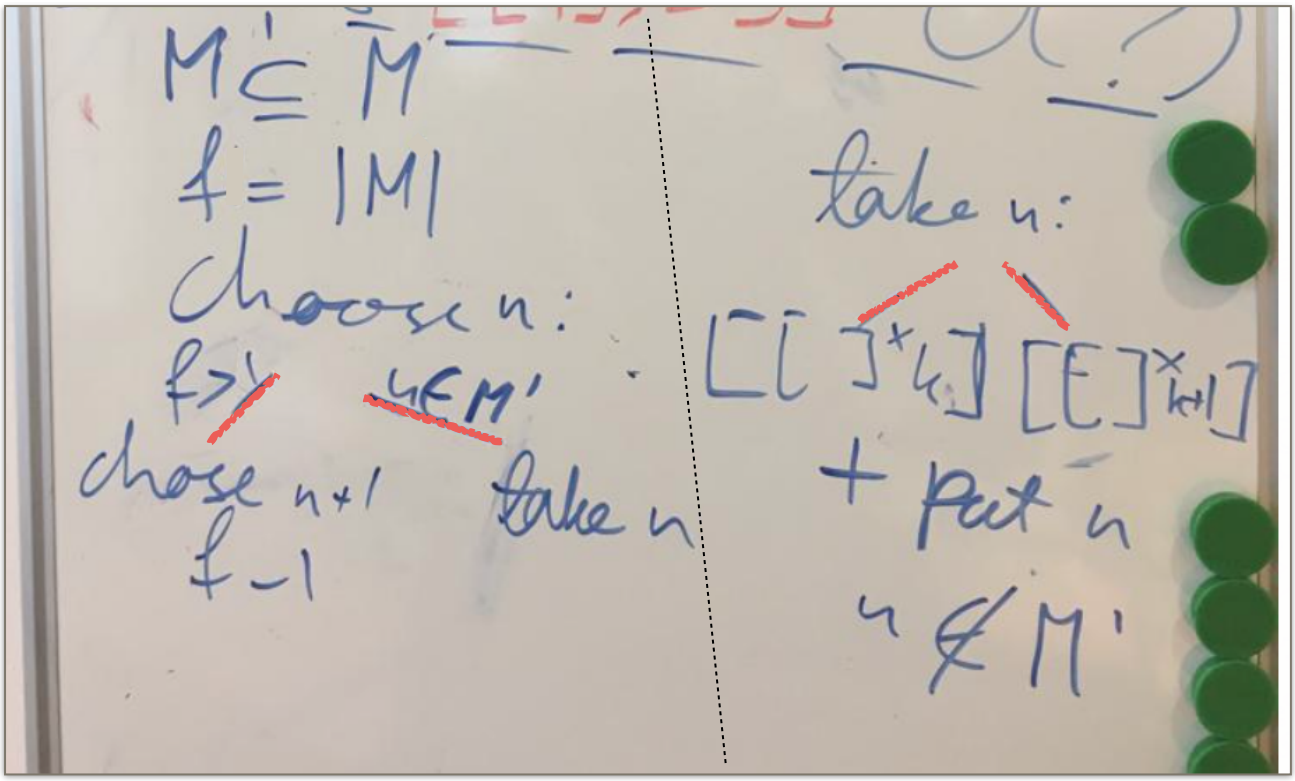
1. Identifizieren des Problems
2. Formulierung des Problems
3. Entwurf eines Algorithmus
   — Falls nötig in Pseudocode
4. Implementierung des Algorithmus und
5. Anwendung des Algorithmus — der Prozess

Lösungsansatz:

FORMULIERUNG:
Diese Aufgabe besteht aus zwei Teilen. Als Erstes gilt es einen Algorithmus zu schreiben, welcher eine gegebene Menge von Schrebergärten so anordnet, dass sie die minimalste Gesamtfläche besitzt. Der zweite Teil besteht aus der grafischen Wiedergabe der errechneten Anordnung.

ENTWURF:
Wir entscheiden uns, die Gärten in einer Hashmap anzugeben. Diese hat einen Index als Schlüssel und eine Breite-mal-Länge als Wert. Man könne die Daten auch anhand eines Arrays angeben, jedoch denken wir, dass Hashtables intuitiver sind. Der Algorithmus des ersten Teils solle im Vergleich zu dem Zweiten deutlich schwerer sein. Wir suchten lange nach einem cleveren Algorithmus, jedoch denken wir, dass Brute-Force die einzig plausible Lösung ist. Das Problem ist, dass der Input zu stark für einen universellen Algorithmus alternieren kann. Also bauen wir einen Prozess, der nach dem Ausschöpfungsprinzip jede mögliche Anordnung probiert und die Beste wiedergibt. Letztlich würden wir für die grafische Ausgabe ein Open-Source Modul benutzen, welches die Rechtecke klar erkennbar ausgibt.
Wir sehen den Gebrauch von Pseudocode hierbei als überflüssig an, jedoch nicht für den Brute-Force Algorithmus. Bei welchem wir uns bei der Komplexität nicht vollständig im klaren sind.

KOMPLEXITÄT VON DEM ALGORITHMUS:
Da das Brute-Force Model exponentielle Zeit beinhaltet, sollte es generell nur als eine Ausweichlösung gelten. Jedoch ist die Größe in unserem Fall nicht höher als 5, weswegen es hier in Ordnung ist. Exponentielle Zeit bedeutet, dass der Exponent der Programmgröße vom Input abhängt, weswegen sie schnell ansteigt.
Wenn *n* Element einer Menge *M* ist, gilt:



Die Menge *M'* ist eine Teilmenge der Ausgangsmenge *M* und fungiert in diesem Fall, als eine Instanz, welche Elemente bereits integriert sind.
f ist die Mächtigkeit der Teilmenge *M.*

DIE STRUKTUR DES BAUMES:
Es werden jeweils zwei Wege eingeschlagen (insofern einige Bedingungen erfüllt sind).
Einerseits wird *n* genommen, insofern *n* noch nicht in der Teilmenge *M'* ist.
Andererseits wird *n* übersprungen und der selbe Zweig wird mit *n+1* aufgerufen, insofern noch mehr als ein Element in der Teilmenge *M'* vorhanden ist.
Falls ein Element genommen wird, werden zwei weitere Wege eingeschlagen. In Einem wird in der gleichen Reihe fortgesetzt und in dem Anderen wird eine Reihe hinzugefügt (*k* ist die Anzahl von Reihen). Letztlich wird *n* dem Zweigresultat hinzugefügt und aus der Teilmenge *M'* ausgeschlossen.

Hinweis:
Die Reihenfolge ist hier leicht verschroben, da *n* zuerst dem Resultat beigefügt werden sollte. Erst darauf solle die Reihenanzahl ermittelt werden.

Umsetzung:

Wir entschieden uns Python als Programmiersprache zu benutzen, da es eine allumfassende Sprache und für ein so primitives Programm mehr als ausreichend ist.

Brute-Force Algorithmus für Anordnung

Bei einem vorgegebenem Hashtable von Gärten gibt diese Funktion die beste Anordnung wieder.

```
63  @timer
64  def fillParts(gardens, memo, gardenIndex=-1, curRow=0, usedGardens=list(), skip=True, arrangement=list() ):
65      if len(usedGardens)>=len(gardens):
66          return arrangement
67      temp1,temp2,temp3,temp4=[[-1]],[[-1]],[[-1]],[[-1]]
68      used=False
69      if curRow>len(arrangement)-1:
70          arrangement.append(list())
71      if not skip:
72          if gardenIndex not in usedGardens:
73              usedGardens.append(gardenIndex)
74              arrangement[curRow].append(gardenIndex)
75              used=True
76      if used:
77          temp1 = fillParts(gardens, memo, 0, curRow, c.deepcopy(usedGardens), False, c.deepcopy(arrangement))
78          temp2 = fillParts(gardens, memo, 0, curRow+1, c.deepcopy(usedGardens), False, c.deepcopy(arrangement))
79      else:
80          gardenIndex += 1
81          if gardenIndex<len(gardens)-1:
82              temp3 = fillParts(gardens, memo, gardenIndex, curRow, c.deepcopy(usedGardens), True, c.deepcopy(arrangement))
83          if gardenIndex in gardens and gardenIndex not in usedGardens:
84              temp4 = fillParts(gardens, memo, gardenIndex, curRow, c.deepcopy(usedGardens), False, c.deepcopy(arrangement))
85      return compare(gardens, memo, temp1, temp2, temp3, temp4)
```

Funktionsweise:

- die Funktion *fillParts* nimmt 5 Parameter:
  - *gardens* ist ein Dictionary mit n-Gärten
  - *memo* ist ebenfalls ein Dictionary jedoch für Memoisierung (bei zugehöriger Funktion erklärt)
    - wird in der *compare*-Funktion benutzt
  - *gardenIndex* ist ein Zähler für den momentanen Garten
  - *curRow* ist ein Zähler für die momentane Reihe
  - *usedGardens* ist ein Array mit benutzten Gärten
  - *skip* ist eine Boolean die evaluiert, ob der momentane Garten in die Anordnung integriert wird
  - *arrangement* ist ein zweidimensionales Array mit der endgültigen Anordnung
- Base-Case —> falls die Mächtigkeit der Anordnungsliste (größer-)gleich die der initialen Liste ist, wird die Anordnung wiedergegeben — Zeile 65
  - die Größer-Gleich-Anweisung ist hier obsolet. Es dient nur der Prävention, falls irgendwie ein ungewollter Wert eingefügt wurde
- Initialisierung von 4 Ästen sowie einer *used* Variable — Zeile 67
- insofern die Anzahl der Reihen inkrementiert wird eine neue Reihe (in Form eines Arrays) hinzugefügt — Zeile 72
- falls momentaner Garten nicht übersprungen werden soll, wird er eingebunden — Zeile 72
  - aktualisiere Liste von benutzten Gärten
  - setze *used* auf wahr
- insofern der Garten genommen wurde, geht der Algorithmus in zwei verschiedene Äste — Zeile 77

  - ein Ast, indem in der selben Reihe fortgesetzt wird
  - und in dem Anderen wird eine Neue hinzugefügt
- wurde der momentane Garten nicht eingebunden, so werden wieder zwei separate Wege eingeschlagen — Zeile 80
  - in einem wird der nächste Garten ebenfalls übersprungen (insofern der nächste nicht der Letzte ist)
  - und in dem Anderen wird der Nächste genommen (insofern der Nächste überhaupt noch in Reichweite ist und noch nicht benutzt wurde)
- letztlich werden alle vier temporären Wege verglichen und der Effektivste wiedergegeben — Zeile 86

Vergleichen von 4 Zweigen

Die *compare*-Funktion nimmt ein Array von Zweigen (in dem Fall 4) und gibt den mit der kleinsten Fläche wieder.

```
47  def compare(initialGardens, memo, *temps):
48      optimal=[[-1]]
49      optArea=sys.maxsize
50      ## memoization -> computation for all of the gardens is linear
51      for gardens in temps:
52          if str(gardens) in memo:
53              gardensArea = memo[str(gardens)]
54          else:
55              bestGardensArrangement = arrange(initialGardens, gardens)
56              gardensArea = getArea(bestGardensArrangement[0], bestGardensArrangement[1])
57              memo[str(gardens)] = gardensArea
58          # update
59          optimal = gardens if (gardensArea < optArea) else optimal
60          optArea = gardensArea if (gardensArea < optArea) else optArea
61      return optimal
```

Funktionsweise:

- Memoisierung — Zeile 51 :
  - falls eine Anordnung von Gärten bereits berechnet, wird sie nun einfach wiedergegeben —> verkürzt die Berechnungszeit enorm, da die Berechnung der Menge aller Anordnungen nun linear ist
- sonst wird für jede Anordnung die kleinstmögliche Fläche errechnet — Zeile 54
- falls die neue Anordnung optimaler ist als das vorherige Optimum, wird es aktualisiert — Zeile 58

Fläche einer Anordnung

Eine Anordnung wird als Parameter eingegeben und die Wiedergabe besteht aus einem Tuple von Breite mal Höhe.

```
10  def arrange(gardens, tempGardens):
11      if tempGardens[0][0]<0: return sys.maxsize, sys.maxsize
12      totalWidth = int()
13      totalHeight = int()
14      matrix = [list() for row in range(len(tempGardens))]
15      for row in range(len(tempGardens)):
16          width=0
17          for col in range(len(tempGardens[row])):
18              height=0
19              startHeight=0
20              addedWidth=width+gardens[tempGardens[row][col]][0]
21              if row>0:
22                  for prevRow in range(row-1, -1, -1):
23                      aboveCol=0
24                      if addedWidth>matrix[prevRow][len(matrix[prevRow])-1][0]:
25                          aboveCol = len(matrix[prevRow])-1
26                      else:
27                          while addedWidth>matrix[prevRow][aboveCol][0] and  aboveCol<len(matrix[prevRow])-1:
28                              aboveCol+=1
29                      while width<matrix[prevRow][aboveCol][0] and aboveCol>0:
30                          aboveHeight=matrix[prevRow][aboveCol][1]
31                          startHeight = aboveHeight if(aboveHeight>startHeight) else startHeight
32                          aboveCol-=1
33                      aboveHeight = matrix[prevRow][aboveCol][1]
34                      startHeight = aboveHeight if(aboveHeight>startHeight) else startHeight
35              height=startHeight+gardens[tempGardens[row][col]][1]
36              width=addedWidth
37              matrix[row].append([width, height])
38              totalWidth = width if(width>totalWidth) else totalWidth
39              totalHeight = height if(height>totalHeight) else totalHeight
40      return totalWidth, totalHeight
```

Funktionsweise:

- falls die Anordnung eine Niete ist — Zeile 11
- für jede Reihe wird eine Breite ausgerechnet und die Größte ausgegeben — Zeile 15
- für jede Spalte wird eine Höhe ausgerechnet und ebenfalls die Größte ausgegeben — Zeile 17
- bei jedem Garten werden die darüber liegenden Gärten überprüft, indem sich die Starthöhe befindet —> damit Gärten sich nicht überlappen

Errechnen einer Fläche

Diese Funktion bedarf keiner weiteren Erklärung. Sie nimmt eine Höhe und eine Breite und multipliziert diese.

```
6   def getArea(width, height):
7       return width*height
```

Errechnen der Koordinaten für die grafische Ausgabe

Diese Funktion ist ziemlich identisch mit der, der Errechnung der Fläche. Jedoch wird in dieser am Ende eine Tuple mit Koordinaten aus jeweiligen Breiten und Höhen ausgegeben.

```
97  def arrangeWithCoords(gardens, tempGardens):
98      if not gardens:
99          return (list(),list())
100     if tempGardens[0][0]<0: return None
101     totalWidth=int()
102     totalHeight=int()
103     startCoords = [list() for row in range(len(tempGardens))]
104     endCoords = [list() for row in range(len(tempGardens))]
105     measure = [[gardens[tempGardens[y][x]] for x in range(len(tempGardens[y]))] for y in range(len(tempGardens))]
106     for row in range(len(tempGardens)):
107         width=0
108         for col in range(len(tempGardens[row])):
109             height=0
110             startHeight=0
111             addedWidth=width+gardens[tempGardens[row][col]][0]
112             if row>0:
113                 for prevRow in range(row-1, -1, -1):
114                     aboveCol=0
115                     if addedWidth>endCoords[prevRow][len(endCoords[prevRow])-1][0]:
116                         aboveCol = len(endCoords[prevRow])-1
117                     else:
118                         while addedWidth>endCoords[prevRow][aboveCol][0] and  aboveCol<len(endCoords[prevRow])-1:
119                             aboveCol+=1
120                     while width<endCoords[prevRow][aboveCol][0] and aboveCol>0:
121                         aboveHeight=endCoords[prevRow][aboveCol][1]
122                         startHeight = aboveHeight if(aboveHeight>startHeight) else startHeiendCoordsght
123                         aboveCol-=1
124                     aboveHeight = endCoords[prevRow][aboveCol][1]
125                     startHeight = aboveHeight if(aboveHeight>startHeight) else startHeight
126             startCoords[row].append([width, startHeight])
127             height=startHeight+gardens[tempGardens[row][col]][1]
128             width=addedWidth
129             endCoords[row].append([width, height])
130             totalWidth = width if(width>totalWidth) else totalWidth
131             totalHeight = height if(height>totalHeight) else totalHeight
132     return (startCoords, endCoords)
```

Grafische Ausgabe

Für die grafische Ausgabe haben wir uns für ein open source Programm von John Zelle entschieden. Dieses ist ein einfaches, benutzerfreundliches, objektorientiertes, python Modul. Es muss sich nur in dem gleichen Ordner befinden.

Ausserdem muss man in seinem Hauptprogramm eine *main*-Funktion haben und die *graphics.py* importieren.

```
38  def main(gardens, scalingFactor=1):
39      win = GraphWin("Allotment Garden", 1000, 1000)
40      win.setBackground(color_rgb(255,255,255))
41
42      ## RECTANGLE
43      result=fillParts(gardens, {})
44      coords=arrangeWithCoords(gardens, result)
45      startCoords = coords[0]
46      endCoords = coords[1]
47      for y in range(len(startCoords)):
48          for x in range(len(startCoords[y])):
49              rect = Rectangle(Point(startCoords[y][x][0]*scalingFactor, startCoords[y][x][1]*scalingFactor)\
50              ,Point(endCoords[y][x][0]*scalingFactor, endCoords[y][x][1]*scalingFactor))
51              rect.setOutline(color_rgb(0, 0, 0))
52              rect.setFill(color_rgb(r.randint(0,255), r.randint(0,255), r.randint(0,255)))
53              rect.draw(win)
54      ## ---------------------------------------- ##
55      win.getMouse()
56      win.close()
57  main(gardens1, 10)
58
```

Funktionsweise:

- die *main*-Funktion nimmt zwei Parameter
  - ein Set von Gartenmaßen
  - und ein Verhältnis
- für das gegebene Set wird die beste Anordnung evaluiert und auf *result* gelegt— Zeile 43
- *coords* ist ein Tuple von Koordinaten aus dem errechneten *result* — Zeile 44
- nun iteriert der Algorithmus durch alle Koordinaten und gibt diese wieder (multipliziert mit dem *scalingFactor*) — Zeile 47

Zeitstopp-Decorator

Dieses separate Programm (*timeThis.py*) beinhaltet nichts, außer einen simplen Decorator welcher

```
63  @timer
64  def fillParts(gardens, memo, gardenIndex=-1, curRow=0, usedGardens=list(), skip=True, arrangement=list() ):
```

die Zeit des Programms stoppt und es in einem Textdokument speichert.
—> natürlich ist diese relativ zur Maschine und dient lediglich dem individuellen Vergleich

```
1  import time
2
3  def timer(method):
4      def inner(*args, **kwargs):
5          startTime=time.time()
6          result = method(*args, **kwargs)
7          endTime=time.time()
8          with open("time.txt", "w") as timef:
9              timef.write("{} took {} seconds to do".format(method.__name__, endTime-startTime))
10         return result
11     return inner
12
```

Beispiele

1.

INPUT SET:

gardens1={
0 : [15, 25],\
1 : [30, 15],\
2 : [25, 15],\
3 : [20, 25]
}

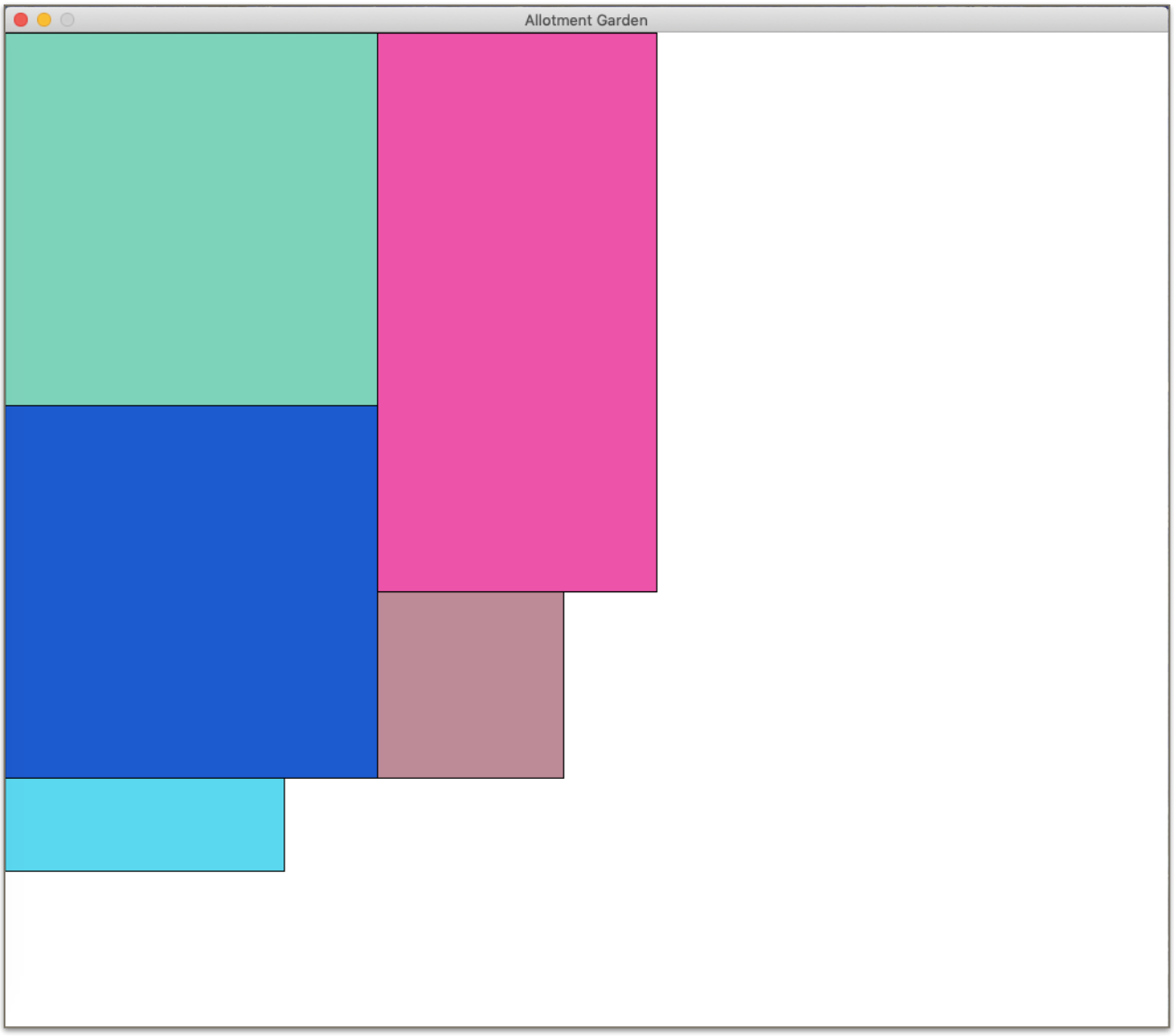
OUTPUT:

2.

INPUT:

```
gardens2={
0 : [3, 6],\
1 : [2, 2],\
2 : [3, 1],\
3 : [4, 4],\
4 : [4, 4]
   }
```

OUTPUT:

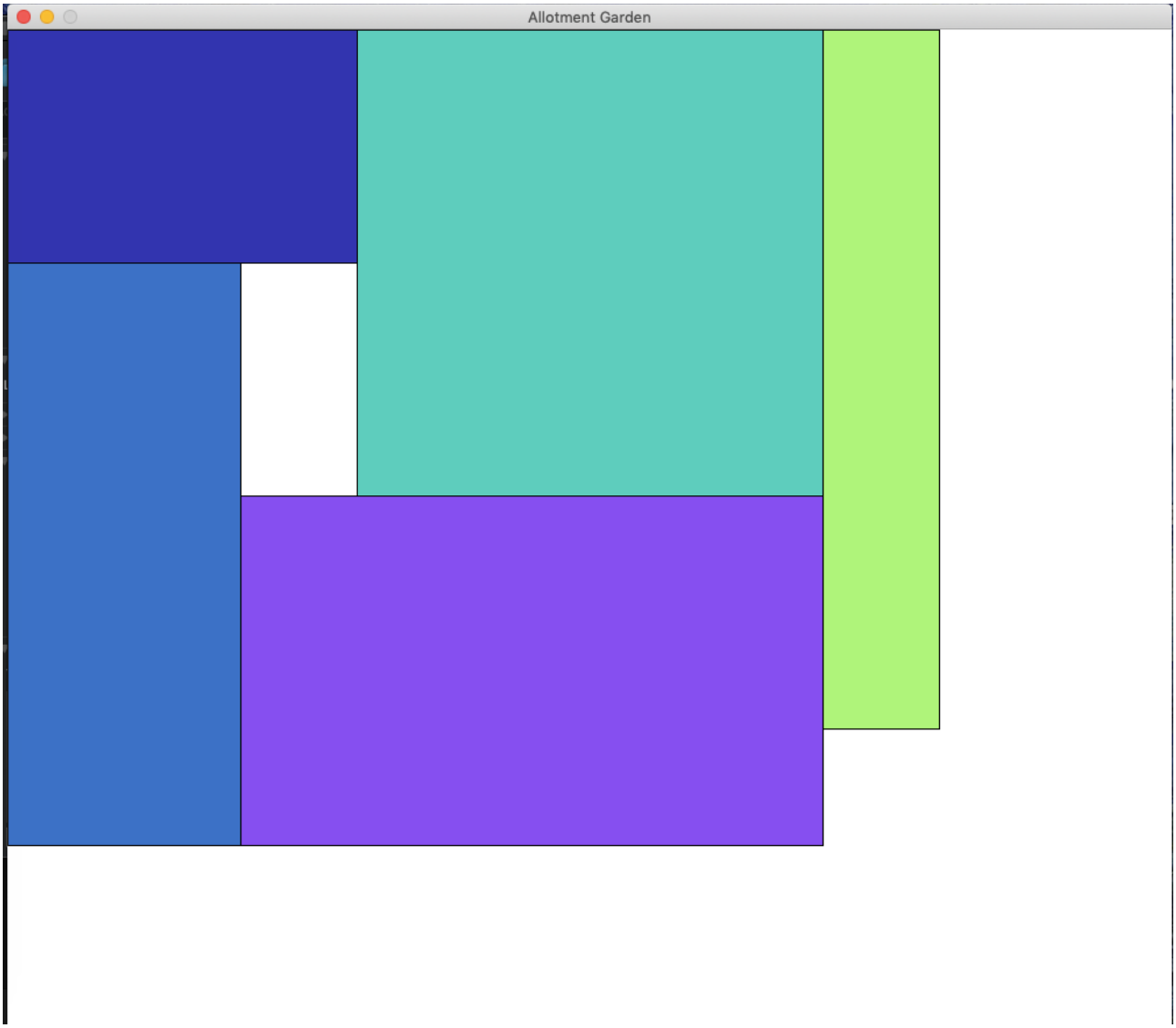


3.

INPUT:

gardens3={
0 : [4, 4],\
1 : [3, 2],\
2 : [1, 6],\
3 : [2, 5],\
4 : [5, 3]
}

OUTPUT:

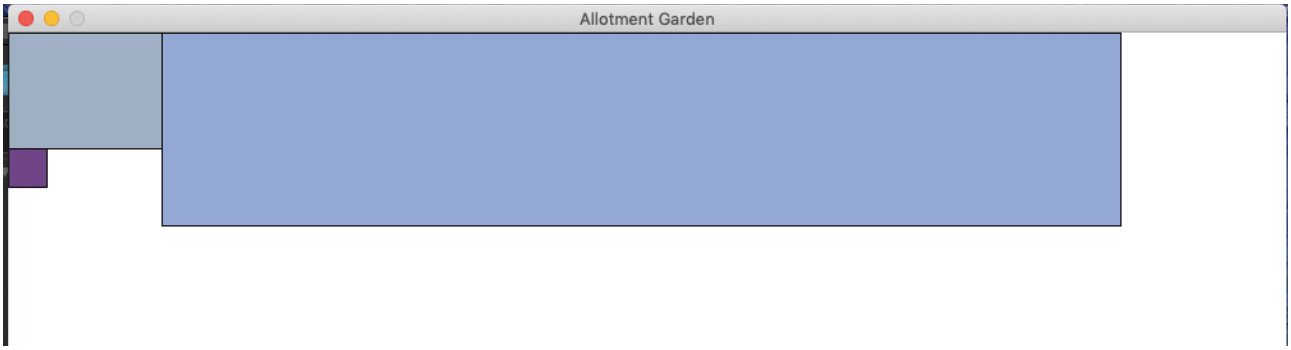


4.

INPUT:

gardens4={
0 : [25, 5],\
1 : [4, 3],\
2 : [1 ,1]
}

OUTPUT:



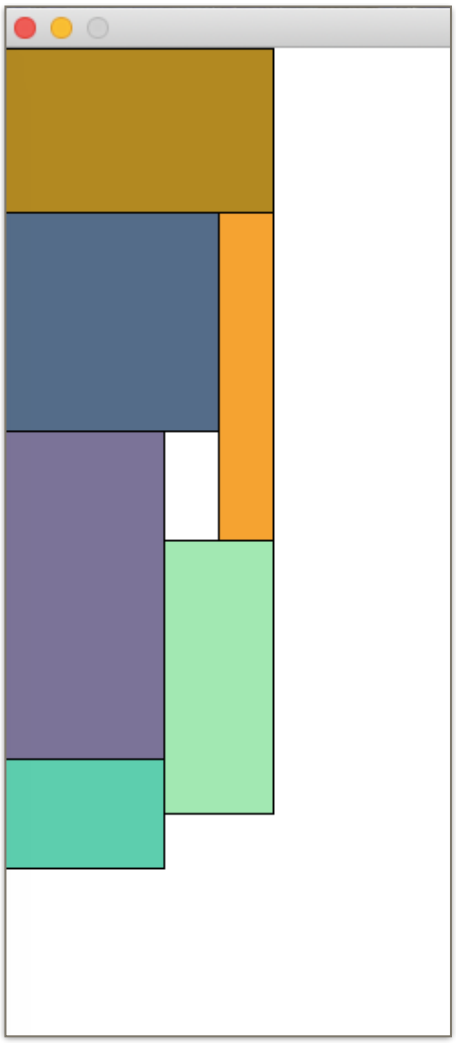
5.

INPUT:

TWISTED:

```
gardens5={
0 : [4, 4],\
1 : [3, 2],\
2 : [1, 6],\
3 : [2, 5],\
4 : [5, 3],\
5: [3,6]
    }
```

OUTPUT:

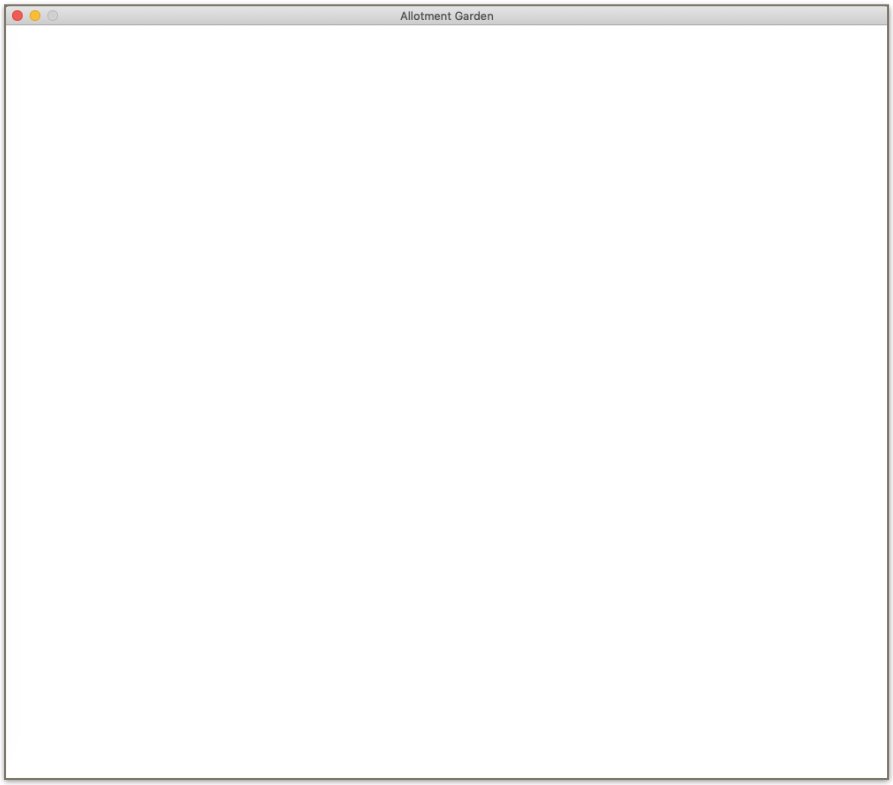


6.

INPUT:

gardens6 = {}

OUTPUT:



Quellcode:

```
import sys
import copy as c
from timeThis import *
counter = int(0)
## return area
def getArea(width, height):
    return width*height

## arranges an set of gardens in the most optimal way
def arrange(gardens, tempGardens):
    if tempGardens[0][0]<0: return sys.maxsize, sys.maxsize
    totalWidth = int()
    totalHeight = int()
    matrix = [list() for row in range(len(tempGardens))]
    # for each row and column get the measure where a garden can be placed
    for row in range(len(tempGardens)):
        width=0
        for col in range(len(tempGardens[row])):
            height=0
            startHeight=0
            addedWidth=width+gardens[tempGardens[row][col]][0]
            if row>0:
                for prevRow in range(row-1, -1, -1):
                    aboveCol=0
                    ##gets upper boundry
                    if addedWidth>matrix[prevRow][len(matrix[prevRow])-1][0]:
                        aboveCol = len(matrix[prevRow])-1
                    else:
                        while addedWidth>matrix[prevRow][aboveCol][0] and
aboveCol<len(matrix[prevRow])-1:
                            aboveCol+=1
                    ## gets lower boudry + sets starting height
                    while width<matrix[prevRow][aboveCol][0] and aboveCol>0:
                        aboveHeight=matrix[prevRow][aboveCol][1]
                        startHeight = aboveHeight if(aboveHeight>startHeight)
else startHeight
                        aboveCol-=1
                    aboveHeight = matrix[prevRow][aboveCol][1]
                    startHeight = aboveHeight if(aboveHeight>startHeight) else
startHeight
            ## update
            height=startHeight+gardens[tempGardens[row][col]][1]
            width=addedWidth
            matrix[row].append([width, height])
            totalWidth = width if(width>totalWidth) else totalWidth
            totalHeight = height if(height>totalHeight) else totalHeight
    return totalWidth, totalHeight
```

```
## out of a possible number of sets picks smallest area
def compare(initialGardens, memo, *temps):
    optimal=[[-1]]
    optArea=sys.maxsize
    ## memoization -> computation for all of the gardens is linear
    for gardens in temps:
        if str(gardens) in memo:
            gardensArea = memo[str(gardens)]
        else:
            bestGardensArrangement = arrange(initialGardens, gardens)
            gardensArea = getArea(bestGardensArrangement[0],
bestGardensArrangement[1])
            memo[str(gardens)] = gardensArea
        # update
        optimal = gardens if (gardensArea < optArea) else optimal
        optArea = gardensArea if (gardensArea < optArea) else optArea
    return optimal
## create a four-branched tree i.e. goes each time in four timelines and picks
best one
@timer
def fillParts(gardens, memo, gardenIndex=-1, curRow=0, usedGardens=list(),
skip=True, arrangement=list() ):
    if len(usedGardens)>=len(gardens):
        return arrangement
    temp1,temp2,temp3,temp4=[[-1]],[[-1]],[[-1]],[[-1]]
    used=False
    if curRow>len(arrangement)-1:
        arrangement.append(list())
    if not skip:
        if gardenIndex not in usedGardens:
            usedGardens.append(gardenIndex)
            arrangement[curRow].append(gardenIndex)
            used=True
    if used:
        temp1 = fillParts(gardens, memo, 0, curRow, c.deepcopy(usedGardens),
False, c.deepcopy(arrangement))
        temp2 = fillParts(gardens, memo, 0, curRow+1, c.deepcopy(usedGardens),
False, c.deepcopy(arrangement))
    else:
        gardenIndex += 1
        if gardenIndex<len(gardens)-1:
            temp3 = fillParts(gardens, memo, gardenIndex, curRow,
c.deepcopy(usedGardens), True, c.deepcopy(arrangement))
        if gardenIndex in gardens and gardenIndex not in usedGardens:
            temp4 = fillParts(gardens, memo, gardenIndex, curRow,
c.deepcopy(usedGardens), False, c.deepcopy(arrangement))
    return compare(gardens, memo, temp1, temp2, temp3, temp4)
```

```
## same method as arrange above, but this one returns the coordinates of the
arrangement
def arrangeWithCoords(gardens, tempGardens):
    if not gardens:
        return (list(),list())
    if tempGardens[0][0]<0: return None
    totalWidth=int()
    totalHeight=int()
    ## for each garden it fills a pair with coordinates (starting- end
coordinates)
    startCoords = [list() for row in range(len(tempGardens))]
    endCoords = [list() for row in range(len(tempGardens))]
    measure = [[gardens[tempGardens[y][x]] for x in range(len(tempGardens[y]))]
for y in range(len(tempGardens))]
    for row in range(len(tempGardens)):
        width=0
        for col in range(len(tempGardens[row])):
            height=0
            startHeight=0
            addedWidth=width+gardens[tempGardens[row][col]][0]
            if row>0:
                for prevRow in range(row-1, -1, -1):
                    aboveCol=0
                    ##gets upper boundry
                    if addedWidth>endCoords[prevRow][len(endCoords[prevRow])-1]
[0]:
                        aboveCol = len(endCoords[prevRow])-1
                    else:
                        while addedWidth>endCoords[prevRow][aboveCol][0] and
aboveCol<len(endCoords[prevRow])-1:
                            aboveCol+=1
                    ## gets lower boundry + sets starting height
                    while width<endCoords[prevRow][aboveCol][0] and aboveCol>0:
                        aboveHeight=endCoords[prevRow][aboveCol][1]
                        startHeight = aboveHeight if(aboveHeight>startHeight)
else startHeight
                        aboveCol-=1
                    aboveHeight = endCoords[prevRow][aboveCol][1]
                    startHeight = aboveHeight if(aboveHeight>startHeight) else
startHeight
            ## update
            startCoords[row].append([width, startHeight])
            height=startHeight+gardens[tempGardens[row][col]][1]
            width=addedWidth
            endCoords[row].append([width, height])
            totalWidth = width if(width>totalWidth) else totalWidth
            totalHeight = height if(height>totalHeight) else totalHeight
    return (startCoords, endCoords)
```