Cost analysis of the algorithm (c_5 → problem_1.py)

```
class Item:
    def __init__(self, value, weight):
        self.value = value                                  #c1=1
        self.weight = weight                                #c2=1

class Node:
    def __init__(self, index, accumulatedValue, accumulatedWeight,
setSelected=None):
        self.index = index                                  #c3=1
        self.accumulatedValue = accumulatedValue            #c4=1
        self.accumulatedWeight = accumulatedWeight          #c5=1
        self.setSelected = setSelected if setSelected
is not None else []                                         #c6=n
        self.upperLimit = 0                                 #c7=1

class PriorityQueue:
    def __init__(self):
        self.data = []                                       #c8=1

    def insert(self, node):
        import heapq                                         #c9=1
        heapq.heappush(self.data, (-node.upperLimit, node)) #c10=log k

    def remover(self):
        import heapq                                            #c11=1
        if not self.data:                                     #c12=1
return None                                                   #c13=1
        return heapq.heappop(self.data)[1]                   #c14=log k

    def isEmpty(self):
        return len(self.data) == 0                          #c15=1

class BranchAndBound:
    def __init__(self, items, W):
        self.items = items                                   #c16=1
        self.W = W                                          #c17=1

    def calcUpperLimit(self, node):
        value = node.accumulatedValue                      #c18=1
        remainWeight = self.W - node.accumulatedWeight      #c19=1
        i = node.index                                         #c20=1
```

```
        n = len(self.items)                                     #c21=1

        while i < n and remainWeight > 0:                      #c22=n
            if self.items[i].weight <= remainWeight:           #c23=n
                value += self.items[i].value                   #c24=n
                remainWeight -= self.items[i].weight           #c25=n
            else:
                fraction = remainWeight / self.items[i].weight #c26=1
                value += self.items[i].value * fraction        #c27=1
                remainWeight = 0                               #c28=1
            i += 1                                              #c29=n

        return value                                           #c30=1

    def KnapsackBandB(self):
        bestValue = 0                                          #c31=1
        bestSet = []                                           #c32=1

        rootNode = Node(index=0, accumulatedValue=0,
accumulatedWeight=0, setSelected=[])                           #c33=n

        rootNode.upperLimit = self.calcUpperLimit(rootNode)    #c34=n

        PQ = PriorityQueue()                                  #c35=1
        PQ.insert(rootNode)                                   #c36=log k

        while not PQ.isEmpty():                               #c37=nlogk
            currentNode = PQ.remover()                        #c38=log k

            if currentNode.upperLimit <= bestValue:           #c39=n
                continue                                      #c40=1

            if currentNode.index == len(self.items):          #c41=nlog k
                if currentNode.accumulatedValue > bestValue:#c42=nlog k
                    bestValue = currentNode.accumulatedValue#c43=nlog k
                    bestSet = currentNode.setSelected         #c44=nlog k
                continue                                      #c45=nlog k

            nextItem = self.items[currentNode.index]          #c46=nlog k

            if currentNode.accumulatedWeight + nextItem.weight <=
self.W:                                                      #c47=nlog k
                includeNode = Node(
```

```
                    index=currentNode.index + 1,
                    accumulatedValue=currentNode.accumulatedValue +
nextItem.value,
                    accumulatedWeight=currentNode.accumulatedWeight +
nextItem.weight,
                    setSelected=currentNode.setSelected +
[currentNode.index]
                )                                         #c48=nlog k
                includeNode.upperLimit =
self.calcUpperLimit(includeNode)                          #c49=nlog k

                if includeNode.upperLimit > bestValue:    #c50=nlog k
                    PQ.insert(includeNode)                #c51=nlog k

            excludeNode = Node(
                index=currentNode.index + 1,
                accumulatedValue=currentNode.accumulatedValue,
                accumulatedWeight=currentNode.accumulatedWeight,
                setSelected=currentNode.setSelected
            )                                             #c52=nlog k
            excludeNode.upperLimit = self.calcUpperLimit(excludeNode)
                                                          #c53=nlog k

            if excludeNode.upperLimit > bestValue:        #c54=nlog k
                PQ.insert(excludeNode)                    #c55=nlog k

        return (bestValue, bestSet)                       #c56=1
```

- Basic operation:
  $c_{36}, c_{37}, c_{38}, c_{41}, c_{42}, c_{43}, c_{44}, c_{45}, c_{46}, c_{47}, c_{48}, c_{49}, c_{50}, c_{51}, c_{52}, c_{53}, c_{54}, c_{55} = n \log k$

- Time complexity calculation:
  $T(n) = (c_{36} + c_{37} + c_{38} + c_{41} + c_{42} + c_{43} + c_{44} + c_{45} + c_{46} + c_{48} + c_{49} + c_{50} + c_{51} + c_{52} + c_{53} + c_{54} + c_{55}) = n \log k$
  $T(n) = 17. n \log k$
  $T(n) = n \log k$
  $T(n) \in O(n \log k)$ $*$ k é o tamanho do heap

- Solving the recurrence:
  $T(n) = n. \log_2 n, \quad n > 0, \quad T(0) = 0$
  $T(1) = 1. \log_2 1 = 1.0 = 0$

$T(2) = 2. log_2 2 = 2. 1 = 2$

$T(3) = 3. log_2 3 = 3. \frac{log3}{log2} = 3. \frac{0.4771}{0.3010} \approx 1. 585 = 3. 1585 \approx 4. 755$

$$T(n) = \sum_{i=1}^{n} log_2 i$$