# 1.

The ADT which fits the desired requirements is AVL tree. This is the tree, where worst case time for all necessary operations is Complexity of all operations is such, because the height of the tree is at each moment.

***LookUp(S, w)*** – is a search operation in AVL tree. Search is performed as a usual search in a binary tree, its complexity is , where is a height of tree.

***Procedure LookUp(S, w):***

*RETURN LookUpStep(S.root, w)*

***Procedure LookUpStep(node, w):*** *//node – is current node*

*IF node == NULL:*

*RETURN NULL*

*IF node.word < w:*

*RETURN LookUpStep(node.right, w)*

*IF node.word > w:*

*RETURN LookUpStep(node.left, w)*

*RETURN node.meanings*

***AddMeaning(S, w, m)*** – is an insert operation in AVL tree. Insertion is made according to the usual binary search tree insertion with necessary rotations on going back to root node after insertion. Since we have to make two traverses (from root to leaf and backwards), complexity is , where is a height of tree.

***DeleteWord(S, w)*** – is a delete operation in AVL tree. Deletion is made according to the usual binary search tree deletion with necessary rotations on going back to root node after deletion. Since we have to make two traverses (from root to leaf and backwards), complexity is , where is a height of tree.

***HasBetween(S, w1, w2)*** – here is the pseudocode of the operation.

***Procedure HasBetween(S, w1, w2):***

*RETURN LookUpStep(S.root, w)*

***Procedure HasBetween(node, w1, w2):*** *//node – is current node*

*IF node == NULL:*

*RETURN FALSE*

*IF node.word =< w1:*

*RETURN HasBetween(node.right, w1, w2)*

*IF node.word >= w2:*

*RETURN HasBetween(node.left, w1, w2)*

*RETURN True*

It is clear, that the complexity of procedure is also , where is a height of the tree: we need only a single traverse from root to leaf.

# 2.

Initially we have 6 variables , each of which is in . And to iterate through all six tuples, we need to generate all tuples of length 6 to filter only tuples, which satisfy:

Where , , , , , , S are given integer constant. We can change given problem to an equivalent problem:

Where new variable are in respective ranges:

It is clear, that for generating all tuples we need time. During the generation we will populate hash table (indeed it is hash map), where stores a list of tuples, such, that . Since, according to requirements insert/search in the hash table will take , it will take totally to populate the hash table in the way described above.

To complete the solution we now will iterate through all tuples . On iterating on each tuple with sum , we will check, if contains any tuples with sum of . If is not empty, each tuple in this collection after concatenation with gives the solution of problem

It is clear, that all solution of transformed problem will be found during such algorithm, since if solution exists, it satisfies:

So algorithm is valid and finds all solutions. Then, one needs to note, that transforming from to can be made in And finally overall complexity is sum of (transforming to ), (building the hash map with all sums of first-half-tuples), (iterating over all second-half-tuples with transformation to and outputting solution). So overall complexity is .

Here is the pseudocode of necessary procedures:

***Procedure GenerateY(N,C,S):*** *// C – array of coefficients: C[1] = A, C[2] = B, etc*

*result := []*

*FOR I FROM 1 TO 6:*

*row := []*

*h := I*

*FOR J FROM 1 TO N:*

*row := row + [h\*C[I] – S/6]*

*h := h\*I*

*result := result + [row]*

*RETURN result*

***Procedure PopulateHashTable(Y,N,C,S):***

*H := new hash map*

*FOR I1 FROM 1 TO N:*

*y1 := Y[1][I1]*

*FOR I2 FROM 1 TO N:*

*y2 := Y[2][I2]*

*FOR I3 FROM 1 TO N:*

*y3 := Y[2][I3]*

*k := y1 + y2 + y3*

*IF H CONTAINS k:*

*H[k] := [(I1, I2, I3)]*

*ELSE:*

*H[k] := H[k] + [(I1, I2, I3)]*

*RETURN H*

***Procedure Solve(N,C,S):***

*Y := GenerateY(N,C,S)*

*H := PopulateHashTable(Y,N,C,S)*

*FOR I4 FROM 1 TO N:*

*y4 := Y[4][I1]*

*FOR I5 FROM 1 TO N:*

*y5 := Y[5][I5]*

*FOR I6 FROM 1 TO N:*

*y6 := Y[6][I6]*

*k := y4 + y5 + y6*

*IF H CONTAINS -k:*

*FOR (I1,I2,I3) IN H[-k]:*

*OUTPUT(I1,I2,I3,I4,I5,I6)*

*RETURN*