Solution — Planks

1 Modeling

The problem states that you are given (one-dimensional) planks of different lengths and you are asked in how many different ways can you create a square by using all of the planks. Let $x_1, \ldots, x_n$ denote the lengths of the n planks. Since the arrangement of the planks within a side of the square does not matter, each of the four sides can be represented by a subset $S_i \subseteq \{x_1, \ldots, x_n\}$, for $i \in \{1, \ldots, 4\}$. Moreover, as one plank can not be used twice and at the same time all the planks must be used we have

$$S_1 \cup S_2 \cup S_3 \cup S_4 = \{x_1, \ldots, x_n\}, \tag{1}$$

for all $1 \leq i < j \leq 4$

$$S_i \cap S_j = \emptyset \tag{2}$$

and for each $k \in \{1, 2, 3, 4\}$

$$\sum_{x' \in S_k} x' = S/4, \tag{3}$$

where $S := x_1 + \ldots + x_n$. The last equation comes from the fact that the total sum of the plank lengths within a side must be equal for each side, as they together form a square. Thus, the number of ways we can partition set $\{x_1, \ldots, x_n\}$ into four pair-wise disjoint sets $\{x_1, \ldots, x_n\} = S_1 \cup S_2 \cup S_3 \cup S_4$ such that (3) holds, denoted by X, is almost what the problem asks from you. The last detail that one needs to notice is that for a given 4-tuple of sets (S_1, S_2, S_3, S_4) that satisfy (1), (2) and (3), any relabeling of the set indices produces the same square. As there are $4! = 24$ different ways to label a 4-tuple, we conclude that the correct output is $X/24$.

2 Algorithm Design

By looking at the constraints we see that n is at most 20. This suggests that a solution running in exponential time might be sufficient. Let us start by analyzing a brute-force approach.

$O(n \cdot 4^n)$ solution (20 points) - In how many ways can we partition a set of n elements into four disjoint subsets? Since each element is a member of one of the four sets, there are exactly 4^n different partitions. Since not all of the partitions satisfy (3) (they satisfy (1) and (2) by definition), we need to discard those that don't. Generating the partitions of $\{x_1, \ldots, x_n\}$ consisting of 4 sets and checking which partition satisfies (3) can be done in $O(n \cdot 4^n)$ time. For $n = 20$ such algorithm would time out, but we can attack the first test set which guarantees $n \leq 8$. Furthermore, since the first two test sets also guarantee that the answer is always 0 or 1, we can output 1 as soon as we find one partition with the desired properties or output 0 if we find no such partition.

$O(n \cdot 2^n)$ solution (60 - 100 points) - Using the *Split and List* technique presented in the tutorial of week 5, one can achieve a much faster running time than in the previous solution. The Split and List technique was illustrated on the problem Subset Sum. In that problem you are given a set X and a number k and your task is to figure out if there exists a subset $S \subseteq X$ such that the sum of the elements in S is equal to k. This problem can be reformulated in a slightly different way: is there a partition of X into two disjoint sets $X = S_1 \cup S_2$ such that

$$\sum_{x' \in S_1} x' = k \quad \text{and} \quad \sum_{x'' \in S_2} x'' = S - k,$$

where S is the total sum of elements in X. This formulation looks quite similar to our current problem and we'll try to use the same strategy as for solving Subset Sum.

Let $X = X_1 \cup X_2$ be an arbitrary partition of set X into two disjoint sets such that $|X_1| = |X_2| = n/2$ (for simplicity we assume n is even). Let $i \in \{1, 2\}$ and let F_i be defined as follows

$$F_i = \{(s_1, s_2, s_3, s_4) : \exists \text{ partition of } X_i \text{ into four disjoint sets } X_i = P_1 \cup P_2 \cup P_3 \cup P_4 \\ \text{such that for all } j \in \{1,2,3,4\} \text{ we have } \sum_{x' \in P_j} x' = s_j\}. \quad (4)$$

Having F_1 and F_2 it is not hard to see that the following algorithm correctly determines if our problem has 0 or more solutions (but not the exact number)

```
int n_solutions = 0
SQ = (x1 + ... + xn) / 4
for any 4-tuple (s1, s2, s3, s4) from F_1:
    if there exist (SQ - s1, SQ - s2, SQ - s3, SQ - s4) in F_2:
      n_solutions = 1
```

Thus if we can test a membership of a 4-tuple in F_2 in time t, then the previous algorithm can be implemented in time roughly equal to $t \cdot |F_1|$. By using an appropriate data structure, e.g. set, testing for membership can be done in time $O(\log |F_2|)$. The size of set F_1 is at most $4^{n/2}$ as there are at most $4^{n/2}$ partitions of F_1 into 4 sets and thus the previous algorithm runs in time

$$4^{n/2} \log 4^{n/2} = n \cdot 2^n.$$

This approach achieves 60 points.

In order to get the exact number of solutions (and consequently full points), we need to modify our algorithm a bit. First, we need that F_2 is stored using a data structure which supports multiple copies of the same element, i.e. vector, map, etc. The next algorithm computes the number of solutions

```
int n_solutions = 0
SQ = (x1 + ... + xn) / 4
for each 4-tuple (s1, s2, s3, s4) in F_1:
    counter = number of copies of (SQ - s1, SQ - s2, SQ - s3, SQ - s4) in F_2
    n_solutions = n_solutions + counter;
n_solutions = n_solutions / 24
```

Although data structure `multiset` from STL (Standard C++ Library) sounds like a natural candidate, we advice *against* using `multiset`. Counting the number of copies of an element in `multiset` runs in linear time in the number of copies of the element. As we would like to execute line 4 in time $\log |F_2|$ this is not good enough. Instead, we can store F_2 in a vector and sort it before iterating through F_1. Now, we can execute execute line 4 in time $O(\log |F_2|)$ by using binary search.

Remark: Although the approach described above correctly implemented scores 100 points, there is a way to easily increase the performance of the algorithm by a factor of roughly two. Because of the symmetry of the problem, we can fill F_1 only with 4-tuples such that x_1 is always in P_1. This cuts the size of F_1 in four. In this way we remove some of the symmetries, so in the end you have to divide `n_solutions` by 6 instead of 24.

3 Implementation

Let us first discuss a possible way to implement the bruteforce solution. Probably the easiest solution is to use a back-track approach. We can use `vector<vector<int> > assignment` to store which elements are assigned to which set, where `assignment[i]` contains all elements which are part of S_i. Having this, the following recursive function returns the number of solutions.

```
int back_track(int id){
  if (id >= n){
    bool ok = true;
    for (int i = 0; i < 4; ++i){
      int sum = 0;
      for (int j = 0; j < assignment[i].size(); ++j)
        sum += planks[assignment[i][j]];

      if (sum != side_length){
        ok  = false;
        break;
      }
    }
    if (ok) return 1;
    else return 0;
  }
  int counter = 0;
  for (int side = 0; side < 4; ++side){
    assignment[side].push_back(id);
    counter += back_track(id + 1);
    assignment[side].pop_back();
  }
  return counter;
}
```

Let us now see how to approach the implementation of $O(n \cdot 2^n)$ solution. We can represent F_1, F_2 as `vector<vector<int> > F1, F2`. Both of the vectors can be filled with a similar recursive procedure as in the back-track solution above. After we fill both vectors we need to sort F_2 which can be done as simple as

```
sort(F2.begin(), F2.end());
```

Finally, we want to check how many copies of a particular 4-tuple is contained in F_2. For this we can use functions `equal_range` (which runs in $O(\log |F_2|)$) and `distance` as follows.

```
// Variable target contains our target 4-tuple
pair<vector<vector<int> >::iterator,vector<vector<int> >::iterator> bounds;
bounds=std::equal_range(F2.begin(), F2.end(), target);
```

```
long long counter = std::distance(bounds.first,bounds.second);
```

4 Appendix

The following code is an implementation of $O(n \cdot 2^n)$ solution explained above.

```
#include <iostream>
#include <vector>
#include <algorithm>

typedef std::vector<std::vector<int> > vecvec;

void back_track(int id, int ubound, vecvec &F, vecvec &assignment,
                const std::vector<int> &planks){
  if (id >= ubound){
    std::vector<int> tuple(4, 0);
    for (int i = 0; i < 4; ++i){
      for (int j = 0; j < assignment[i].size(); ++j)
        tuple[i] += planks[assignment[i][j]];
    }
    F.push_back(tuple);
    return;
  }

  for (int i = 0; i < 4; ++i){
    assignment[i].push_back(id); // Try to put id-th plank to i-th set
    back_track(id + 1, ubound, F, assignment, planks); // Recurse
    assignment[i].pop_back(); // Remove id-th plank from i-th set
  }
}

void do_test(int n, const std::vector<int> &planks) {
  int sum = 0;
  long long n_sol = 0;

  for (int i = 0 ; i < planks.size(); ++i)
    sum += planks[i];

  // If the total sum of lengths is not divisible by four,
  // it is not possible to create a square.
  if (sum % 4 != 0){
    std::cout << 0 << std::endl;
    return;
  }
  vecvec F1, assignment(4);
  back_track(0, n/2, F1, assignment, planks); // Generate all 4-tuples
                                              // for the first half of the set.

  vecvec F2, assignment2(4);
  back_track(n/2, n, F2, assignment2, planks); // Generate all 4-tuples
                                               // for the second half of the set.
  sort(F2.begin(), F2.end());

  for (int idx = 0; idx < F1.size(); ++idx){
    std::vector<int> member = F1[idx];
    for (int i = 0; i < 4; ++i)
      member[i] = sum/4 - member[i];
```

```cpp

    std::pair<vecvec::iterator,vecvec::iterator> bounds;
    bounds=std::equal_range(F2.begin(), F2.end(), member);
    long long counter = std::distance(bounds.first,bounds.second);
    n_sol += counter;
  }
  std::cout << n_sol  / 24 << std::endl;
}

int main() {
  int t;
  std::cin >> t;
  while (t--) {
    int n;
    std::cin >> n;
    std::vector<int> planks;
    for (int i = 0; i < n; ++i){
      int plank; std::cin >> plank;
      planks.push_back(plank);
    }
    do_test(n, planks);
  }
}
```