Question 1. We solve the question by dividing the problem to subproblems both in DP paradigm and divide-and-conquer algorithm. What's the difference between them?

Both Dynamic Programming (DP) and Divide-and-Conquer solve problems by breaking them down into subproblems. However, they are fundamentally different in how they reuse and combine these subproblems.

Aspect	Dynamic Programming	Divide and Conquer
Overlapping Subproblems	Yes, many subproblems are reused.	No, subproblems are usually independent.
Optimal Substructure	Required	Required
Memoization / Tabulation	Used to avoid recomputation	Not typically used
Time Efficiency	More efficient due to caching	Can be inefficient if subproblems repeat
Common Examples	Fibonacci, Knapsack, Shortest Path	Merge Sort, Quick Sort, Binary Search

Summary: Dynamic Programming is suitable for problems with overlapping subproblems and optimal substructure. It saves computation time by caching intermediate results. In contrast, Divide-and-Conquer works best when subproblems are independent and doesn't require memoization.

Question 2. In basketball, each successful shot gains either 2 or 3 points. Please write a program to calculate how many distinct ways that a team can get exactly n points.

```
#include <iostream>
#include <vector>
using namespace std;
#define tp top_down
#define bu bottom_up

int tp(int n) {
  static vector<int> dp(1, -1);
  if (n < 0)
    return 0;
  if (n == 0)
    return 1;
  if (n >= dp.size())
    dp.resize(n + 1, -1);
  if (dp[n] != -1)
    return dp[n];

  return dp[n] = tp(n - 2) + tp(n - 3);
}

int bu(int n) {
  vector<int> dp(n + 1, 0);
  for (int i = 1; i <= n; ++i) {
    if (i >= 2)
      dp[i] += dp[i - 2];
    if (i >= 3)
      dp[i] += dp[i - 3];
  }
  return dp[n];
}

int main() {
  int n;
  cin >> n;
  cout << tp(n);
  cout << '\n';
  cout << bu(n);
  return 0;
}
```

Question 3. Some of the stairs were broken. We can not step on the broken stairs. Given the broken stairs by $B = \{b_1, b_2, \ldots, b_m\}, 1 \leq b_1 < b_2 < \cdots < b_m \leq n$, how many ways can we climb to total n stairs?

```
#include <iostream>
#include <unordered_set>
#include <vector>
using namespace std;
#define tp top_down
#define bu bottom_up

unordered_set<int> broken;

int tp(int n) {
  static vector<int> dp = {-1};
  if (n < 0)
    return 0;
  if (broken.count(n))
    return 0;
  if (n == 0)
    return 1;
  if (n >= dp.size())
    dp.resize(n + 1, -1);
  if (dp[n] != -1)
    return dp[n];

  return dp[n] = tp(n - 1) + tp(n - 2);
}

int bu(int n) {
  vector<int> dp(n + 1, 0);
  dp[0] = 1;

  for (int i = 1; i <= n; ++i) {
    if (broken.count(i))
      continue;
    if (i >= 1)
      dp[i] += dp[i - 1];
    if (i >= 2)
      dp[i] += dp[i - 2];
  }
  return dp[n];
}

int main() {
  int n;
  cin >> n;

  broken = {3, 5};

  cout << tp(n);
  cout << '\n';
  cout << bu(n);
  return 0;
}
```

Question 4. Will sorting C by increasing order improve the performance or not? Why?

Sorting the coin set C in increasing order does not improve the algorithm's time complexity, nor does it affect the correctness, because the dynamic programming transition logic depends only on whether we process each coin exactly once in a fixed order.

Question 5. Since $f(C_k, *)$ only depends on $f(C_k, *)$ and $f(C_{k-1}, *)$, we can cache the data in $2 \times m$. The skill named rolling. Please rewrite above algorithm to a rolling table version.

```
int coin_change_rolling(const std::vector<int>& coins, int target) {
```

```cpp
  int n = coins.size();
  std::vector<std::vector<int>> dp(2, std::vector<int>(target + 1, 0));
  dp[0][0] = 1;

  for (int k = 0; k < n; ++k) {
    int cur = k % 2;
    int prev = (k + 1) % 2;

    for (int amt = 0; amt <= target; ++amt) {
      dp[cur][amt] = dp[prev][amt];
      if (amt >= coins[k])
        dp[cur][amt] += dp[cur][amt - coins[k]];
    }
  }

  return dp[(n - 1) % 2][target];
}
```