main.cpp

```cpp
#include <iostream>
#include <vector>

using namespace std;

struct Data {
  int value = 0;
  bool direction = true; // true for <-- || false for -->
  Data(int);
};

Data::Data(int val) {
  this->value = val;
}

float Total(float); // function for finding total permutations, 25 too big to be represented using int, so used float
void get_Set(ostream&, vector<Data>); // prints the passed in permutation
int findLargestMobile(vector<Data>&); // finds largest mobile element k
void Johnson_Trotter(vector<Data>&, vector<vector<Data> >&);    // follows Johnson-Trotter algo to manipulate given vector, adds to list of permutations

int main() {
  cout << "1 = <--    0 = -->" << endl;
  int x = 0;
  // get user input for size
  cout << "Enter a positive integer between 1 and 25: " << endl;
  cin >> x;
  while (x < 1 || x > 25) {
    cout << "Input is not within the range of 1-25." << endl;
    cin >> x;
  }

  float size = x; // use numbers 1 to size

  vector<Data>Numbers;
  vector<vector<Data>> Permutations;

  for (int i = 1; i <= size; i++) { // initializes initial permutation
```

```
    Numbers.push_back(Data(i));
  }

  cout << "There are " << Total(size) << " Permutations of the set ";
  get_Set(cout, Numbers);
  cout << ":\n";
  Johnson_Trotter(Numbers, Permutations);
  cout << "\nPermutations Stored: " << Permutations.size() << endl;
  return 0;
}

float Total(float size) {
  float Permutations = 1;
  while (size != 0) {
    Permutations = Permutations * size;
    size--;
  }
  return Permutations;
}

void get_Set(ostream& out, vector<Data> Numbers) {
  out << "{";
  for (int i = 0; i < Numbers.size() - 1; i++) {
    out << Numbers[i].value << ", ";
  }
  out << Numbers[Numbers.size() - 1].value << "}";
}

int findLargestMobile(vector<Data>& Numbers) {
  int largest = -1;        // if -1 returned, found all permutations
  int valueOfLargest = 0;
  int index = 1;          // start search at index 1 since special rule for index 0
  int next = index + 1;   // next points to element after index, when next is == size, we
know index points to final element. Final element has special rules so need to do
seperate from while loop

  if (!Numbers[0].direction) { // first element points right, compare with next element. If
points left, can't be mobile
    if (Numbers[0].value > Numbers[index].value) {
      largest = 0;
      valueOfLargest = Numbers[largest].value;
    }
  }

  while (next != Numbers.size()) {
    if (Numbers[index].direction) {     // current element points left
      if (Numbers[index].value > Numbers[index - 1].value) {
        if (Numbers[index].value > valueOfLargest) {
          largest = index;
          valueOfLargest = Numbers[largest].value;
        }
      }
    }
    else {                              // current element points right
      if (Numbers[index].value > Numbers[next].value) {
        if (Numbers[index].value > valueOfLargest) {
          largest = index;
          valueOfLargest = Numbers[largest].value;
        }
      }
    }
    index++;
    next++;
  }

  if (Numbers[index].direction) {    // last element points left, compare with prev element.
If points right, can't be mobile
    if (Numbers[index].value > Numbers[index - 1].value) {
      if (Numbers[index].value > valueOfLargest) {
        largest = index;
        valueOfLargest = Numbers[largest].value;
      }
    }
  }
  return largest;
}

void Johnson_Trotter(vector<Data>& Numbers, vector<vector<Data>>& Permutations) {
  get_Set(cout, Numbers);
  Permutations.push_back(Numbers);
  cout << "\n";

  if (Numbers.size() == 1) { // single element contained in Numbers, all permutations just single element
    return;
  }

  int indexOfLargestMobile = 0;
  int valueOfLargestMobile = 0;
  bool directionOfLargestMobile = true;
  while (true) {
    indexOfLargestMobile = findLargestMobile(Numbers);    // find largest mobile element

    if (indexOfLargestMobile == -1) {                    // if return -1, found all permutations
      return;
    }

    valueOfLargestMobile = Numbers[indexOfLargestMobile].value;      // Store the value of the largest mobile element
    directionOfLargestMobile = Numbers[indexOfLargestMobile].direction;     // Store the direction of the largest mobile element

      // Swap the largest mobile element with the adjacent element
    if (directionOfLargestMobile) { // Pointing left
      swap(Numbers[indexOfLargestMobile], Numbers[indexOfLargestMobile - 1]);
    }
    else { // Pointing right
      swap(Numbers[indexOfLargestMobile], Numbers[indexOfLargestMobile + 1]);
    }

    // Update the directions of elements larger than the largest mobile element
    for (int i = 0; i < Numbers.size(); i++) {
      if (Numbers[i].value > valueOfLargestMobile) {
        Numbers[i].direction = !Numbers[i].direction; // Reverse direction
      }
    }
    Permutations.push_back(Numbers);
    get_Set(cout, Numbers);
    cout << "\n";
  }
```

```
}
```

TestDriver.cpp

```
/*************************************************************************************
*    Title: C++ Plus Data Structures SIXTH EDITION
*    Author: Nell Dale, Chip Weems
*    Date: 08/30/2023
*    Code version: C++11
*    Availability: Pages 138-139
*
*************************************************************************************/

#include <iostream>
#include <fstream>
#include <string>
#include <cstring> // stof
#include <vector>

using namespace std;

struct Data {
  int value = 0;
  bool direction = true; // true for <-- || false for -->
  Data(int);
};

Data::Data(int val) {
  this->value = val;
}

float Total(float); // function for finding total permutations, 25 too big to be represented
using int, so used float
void get_Set(ostream&, vector<Data>); // prints the passed in permutation
int findLargestMobile(vector<Data>&); // finds largest mobile element k
void Johnson_Trotter(vector<Data>&, vector<vector<Data> >&);    // follows
Johnson-Trotter algo to manipulate given vector, adds to list of permutations

int main()
{
  ifstream inFile;     // file containing operations
```

```
ofstream outFile;    // file containing output
string inFileName;   // input file external name
string outFileName;  // output file external name
string outputLabel;
string command;    // # to use for size
int numCommands;
float size = 0; // use numbers 1 to size
vector<Data>Numbers;
vector<vector<Data>> Permutations;

// input file
inFile.open("Test.txt");

// output file
outFile.open("Results.txt");

cout << "Enter name of test run; press return." << endl;
cin >> outputLabel;
outFile << outputLabel << endl;

inFile >> command;
numCommands = 0;
while (command != "Quit") {
  if (isdigit(command[0])) {
    size = stof(command,NULL);
    if (size > 0 && size <= 25) {
      for (int i = 1; i <= size; i++) { // initializes initial permutation
        Numbers.push_back(Data(i));
      }
      outFile << "There are " << Total(size) << " Permutations of the set ";
      get_Set(outFile, Numbers);
      Johnson_Trotter(Numbers, Permutations);
      outFile << "\nPermutations Stored: " << Permutations.size() << endl;
      Permutations.clear();
      Numbers.clear();
    }
  }
  else {
    outFile << command << " IS AN INVALID NUMBER/COMMAND" << endl;
  }
```

```
    numCommands++;
    cout << "Command number " << numCommands << " completed."
      << endl;
    inFile >> command;
  }
  cout << "Testing completed." << endl;
  inFile.close();
  outFile.close();
  return 0;
}


float Total(float size) {
  float Permutations = 1;
  while (size != 0) {
    Permutations = Permutations * size;
    size--;
  }
  return Permutations;
}

void get_Set(ostream& out, vector<Data> Numbers) {
  out << "{";
  for (int i = 0; i < Numbers.size() - 1; i++) {
    out << Numbers[i].value << ", ";
  }
  out << Numbers[Numbers.size() - 1].value << "}";
}

int findLargestMobile(vector<Data>& Numbers) {
  int largest = -1;        // if -1 returned, found all permutations
  int valueOfLargest = 0;
  int index = 1;          // start search at index 1 since special rule for index 0
  int next = index + 1;   // next points to element after index, when next is == size, we
know index points to final element. Final element has special rules so need to do
seperate from while loop

  if (!Numbers[0].direction) { // first element points right, compare with next element. If
points left, can't be mobile
    if (Numbers[0].value > Numbers[index].value) {
      largest = 0;
      valueOfLargest = Numbers[largest].value;
    }
  }

  while (next != Numbers.size()) {
    if (Numbers[index].direction) {     // current element points left
      if (Numbers[index].value > Numbers[index - 1].value) {
        if (Numbers[index].value > valueOfLargest) {
          largest = index;
          valueOfLargest = Numbers[largest].value;
        }
      }
    }
    else {                              // current element points right
      if (Numbers[index].value > Numbers[next].value) {
        if (Numbers[index].value > valueOfLargest) {
          largest = index;
          valueOfLargest = Numbers[largest].value;
        }
      }
    }
    index++;
    next++;
  }

  if (Numbers[index].direction) {    // last element points left, compare with prev element.
If points right, can't be mobile
    if (Numbers[index].value > Numbers[index - 1].value) {
      if (Numbers[index].value > valueOfLargest) {
        largest = index;
        valueOfLargest = Numbers[largest].value;
      }
    }
  }
  return largest;
}

void Johnson_Trotter(vector<Data>& Numbers, vector<vector<Data>>& Permutations) {
  //get_Set(cout, Numbers);
```

```
 Permutations.push_back(Numbers);
 //cout << "\n";

 if (Numbers.size() == 1) { // single element contained in Numbers, all permutations just
single element
  return;
 }

 int indexOfLargestMobile = 0;
 int valueOfLargestMobile = 0;
 bool directionOfLargestMobile = true;
 while (true) {
  indexOfLargestMobile = findLargestMobile(Numbers);    // find largest mobile element

  if (indexOfLargestMobile == -1) {                     // if return -1, found all permutations
   return;
  }

  valueOfLargestMobile = Numbers[indexOfLargestMobile].value;      // Store the value
of the largest mobile element
  directionOfLargestMobile = Numbers[indexOfLargestMobile].direction;     // Store the
direction of the largest mobile element

   // Swap the largest mobile element with the adjacent element
  if (directionOfLargestMobile) { // Pointing left
   swap(Numbers[indexOfLargestMobile], Numbers[indexOfLargestMobile - 1]);
  }
  else { // Pointing right
   swap(Numbers[indexOfLargestMobile], Numbers[indexOfLargestMobile + 1]);
  }

  // Update the directions of elements larger than the largest mobile element
  for (int i = 0; i < Numbers.size(); i++) {
   if (Numbers[i].value > valueOfLargestMobile) {
    Numbers[i].direction = !Numbers[i].direction; // Reverse direction
   }
  }
  Permutations.push_back(Numbers);
  //get_Set(cout, Numbers);
  //cout << "\n";
  }
}
```