A detailed explanation of the code has been included in the code using comments.

What algorithms did I use?
For this code, the algorithm that was used is the quicksort algorithm. This was pretty much required since it was a requirement for this task.

Why this algorithm?
The advantage of using this approach is that we can organize a list in an efficient fast way. Since we are splitting the list into two halves. On the left side resides the lower than the key values and on the other side the larger values. It allows us to sort a list faster, especially on a larger scale.

What other algorithms could I have used here?
Other options that could have been utilized here are other examples we took from the labs, such as insertion, bubble, or selection sort. However, they would have been a less efficient approach.

What are the differences?
The difference is that in bubble sort, for example, we compare adjacent values and swap them if they are not in the right order. The selection sort on the other hand is similar to the quicksort in the way it splits the list into two parts. One part is the sorted elements starting from the left side and the other part is still the remaining unsorted. And finally, the insertion sort sorts the elements of a list one by one after the first element of a list.

Complexity analysis?
The worst-case time complexity of this program is O(n^2). This is because in the worst case, our list may already be sorted or the largest or smallest element of the list is chosen as the pivot. In our case also our pivot in this code is always the last element of the list. Finally, we recursively call our code and navigate through either the entire list or sub-lists. For the future, we could maybe use the median of the list as our pivot but in general, for a doubly linked list, it is quite challenging to make optimization changes to reduce our worst-case scenario.

Code:

```
#include <iostream>
using namespace std;

// Declaring and defining the node class so we can later use it throughout the code
class Node {
public:

   // Value of the nodes
   int value;
   // Pointers to our next and previous nodes this will help us navigate our list
   Node* next;
   Node* prev;
   Node(int val = 0) : value(val), next(nullptr), prev(nullptr) {
   }
};

// Declaring our class for the doubly linked list
class DoublyLinked {
public:
   // Pointer to the first node
   Node* head;
   // Pointer to the last node
   Node* tail;

   DoublyLinked() : head(nullptr), tail(nullptr) {}
```

```
    // Function to insert our newNode at the end of the list
    void insert(int val) {
        // Createing a newNode
        Node* newNode = new Node(val);
        if (head == nullptr) {
            // In case the list was empty we make the newNode both the head and the tail
            head = newNode;
            tail = newNode;
        }
        else {
            // If the list was not empty we will add the newNode to the end of the list and make it the
new tail
            tail->next = newNode;
            newNode->prev = tail;
            tail = newNode;
        }
    }

    // Printing out all the nodes values
    void print() {
        Node* temp = head;
        while (temp != nullptr) {
            cout << temp->value << " ";
            temp = temp->next;
        }
        cout << endl;
    }
};

// Function to swap the values of two nodes
void swapNodes(Node* a, Node* b) {
    int temp = a->value;

    a->value = b->value;

    b->value = temp;

}


// This is our partition function

Node* partition(Node* low, Node* high) {

    // We will be using the last node of the list as our pivot

    int pivot = high->value;

    Node* i = low->prev;


    // Now using our last node as the pivot we will sort the list based off if they are higher or lower
value than the pivots value

    // We will now loop through each node starting from the first node except our pivot which is our
last node

    for (Node* j = low; j != high; j = j->next) {

        // If the current nodes value is less than the pivot value we will move the lower half pivot one
step to the right while ensuring that our i is not at the beginning of the list

        if (j->value < pivot) {

            i = (i == nullptr) ? low : i->next;

            // Lastly, we will swap our nodes values to move all the values that are lower than the pivot to
the lower half side of the list

            swapNodes(i, j);

        }

    }

    // Here we will move our i which is where our pivot needs to be placed at the correct position and
if it is at the beginning of the list we will move to the next node right after

    i = (i == nullptr) ? low : i->next;


    // The last step is to now move our pivot which is still at the end of the list to its intended position
so it's between the lower and higher halfs of the list

    swapNodes(i, high);

    // Returning the pivot position
    return i;
}

// QuickSort algorithm from the lab
void quickSort(Node* low, Node* high) {
    if (high != nullptr && low != high && low != high->next) {
        // Get the pivot position
        Node* pivot = partition(low, high);
        // Recursively calling to sort all values less than the pivot
        quickSort(low, pivot->prev);
        // Recursively calling to sort all values more than the pivot
        quickSort(pivot->next, high);
    }
}

// Main function
int main() {

    // Declaring our doublylinkedlist
    DoublyLinked  list;

    // Dynamic insertion of our nodes
    list.insert(9);
    list.insert(2);
    list.insert(5);
    list.insert(27);
    list.insert(11);
    list.insert(3);
    list.insert(7);
    list.insert(13);
    list.insert(4);
    cout << "Original list: ";
    list.print();

    // Finally we will sort our doublylinkedlist
    quickSort(list.head, list.tail);

    cout << "List after sorting: ";
    list.print();

    return 0;
}
```