Tongji University

Data Structure Course Curriculum Design Document

Task10: Comparative case of eight sorting algorithms

Author:
嘉杰 李
Jiajie Li

Supervisor:
颖 张
Prof. Ying Zhang

A companion use documentation for a program submitted in fulfillment of the requirements for the Data Structure curriculum design

in the

School of Software Engineering, Tongji

December 27, 2019

"Thanks to my solid academic training, today I can write hundreds of words on virtually any topic without possessing a shred of information, which is how I got a good job in journalism."

Dave Barry

Contents

Chapter 1

Analysis

1.1 Background Analysis

The so-called sorting is the operation of making a series of records increase or decrease according to the size of one or some keywords. Sorting algorithm is how to make records arrange according to requirements. Sorting algorithms have received considerable attention in many fields, especially in the processing of large amounts of data. A good algorithm can save a lot of resources. Considering the various restrictions and specifications of data in various fields, to obtain an excellent algorithm that conforms to the actual situation, a lot of reasoning and analysis are required. Therefore, the study of the ranking algorithm is an important part of our learning data structure.

1.2 Functional Analysis

This question requires the user to give the number of random numbers generated, and the system calls a random function to generate one hundred, one thousand, ten thousand, or one hundred thousand random numbers. Then the user enters the corresponding number to choose which operation to use. The numbers 1 to 8 indicate that different sort functions are used for sorting, and the number 9 indicates exiting the program. After the numbers 1 to 8 are entered, the time it takes for the corresponding sorting algorithm to sort and the number of comparisons after sorting are output.

Chapter 2

Design

2.1 Data structure design

As mentioned in the functional analysis above, this program requires a large number of exchange and comparison operations. For convenience, a data table class is used to store the generated random numbers, and 1 to 8 sorting algorithms are encapsulated and called. The random number generation function Also encapsulated in the data table class.

2.2 Class structure design

Since the performance of 8 sorting algorithms need to be compared, the data sorted by each sorting algorithm needs to be the same. Therefore, two arrays are used to store the generated random numbers, one array is used for sorting operations, and one array is used to store the original random Number and used to restore the array used for sorting. Eight sorting algorithms are encapsulated in the data table class, and the corresponding comparison times are returned by them.

2.3 Member function design

Vector Class:

```
template <class T>
class Vector {
public:
    Vector(int size=0):theSize(size),theCapacity(size+SPACE_CAPACITY){ data = new
        T[theCapacity];} // 构造函数
    ~Vector(void) { delete[] data;} // 析构函数
    Vector& operator=(Vector& other){
       // 判断是否为自身赋值
       if (this == &other)
          return *this;
       else {
          delete[]data;
```

```
        theSize = other.theSize;
        theCapacity = other.theCapacity;
        this->data = new T[theCapacity];

        for (int i = 0; i < theSize; ++i) {
            data[i] = other.data[i];
        }
        return *this;
    }
} // 重载赋值函数
Vector(Vector& other) :theSize(0),theCapacity(0),data(NULL){ *this=other; }//
    复制构造函数

// 重新分配空间大小函数
void reServe(int newCapacity) {
    if (newCapacity <= theCapacity)
        return;

    T *temp = data;
    data = new T[newCapacity];
    theCapacity=newCapacity;
    //std::cout << newCapacity << std::endl;
    for (int i = 0; i < theSize; ++i)
        data[i] = temp[i];
    delete[] temp;
}
// push_back函数
void push_back(T val) {
    if (theSize == theCapacity)
        reServe(2 * theCapacity + 1);
    data[theSize++] = val;
}
bool pop_back(){
    if (!theSize) {
        std::cerr << "Fail to pop back, the vector is empty!\n";
        return false;
    }
    theSize--;
    return true;
}
bool Delete(int ind){
    if (ind<0||ind>=theSize){
        std::cerr << "Fail to delete, the ind is out of range!\n";
        return false;
    }
    for (int i=ind;i<theSize-1;++i) data[i]=data[i+1];
    return pop_back();
}
```

```
        T& operator[] (int n) {return data[n];}
        T *data;
        int size() const {return theSize;}
    private:
        const int SPACE_CAPACITY = 16;
        int theCapacity;
        int theSize;
    };
```

Sort Class:

```
    void swap(long &x,long &y){
    long temp;
    temp=x;
    x=y;
    y=temp;
}

int qsort(Vector<long>& v, int L, int R){
    int i=L, j=R, x=v[L];
    long long count=0;
    if (L>=R) return 0;
    while (i<j) {
        while (i<j&&v[j]>=x) --j;
        if (i<j) v[i++]=v[j],++count;
        while (i<j&&v[i]<x) ++i;
        if (i<j) v[j--]=v[i],++count;
    }
    v[i]=x; ++count;
    count+=qsort(v,L,i-1);
    count+=qsort(v,i+1,R);
    return count;
}

int Quick_Sort(Vector<long>& v){
    int len=v.size();
    long long count=0;
    count+=qsort(v,0,len-1);
    return count;
}

int Bubble_Sort(Vector<long>& v){
    int len=v.size();
    long long count=0;
    for (int i=0;i<len-1;++i)
        for (int j=0;j<len-i-1;++j) {
            if (v[j] > v[j + 1]) swap(v[j], v[j + 1]);
            count++;
        }
```

```
    return count;
}

int Selection_Sort(Vector<long>& v){
    int len=v.size();
    long long count=0;
    for (int i=0;i<len;++i) {
        int p=0;
        for (int j = i+1; j < len; ++j) {
            count++;
            if (v[j] < v[p]) p = j;
        }
        swap(v[i],v[p]);
    }
    return count;
}

int Insertion_Sort(Vector<long>& v){
    int len=v.size();
    long long count=0;
    for (int i=0;i<len;++i) {
        int temp=v[i];
        int j=i;
        while (j>=0) {
            count++;
            if (j>0&&v[j-1]>temp) v[j]=v[j-1];
            else {v[j]=temp; break;}
            j--;
        }
    }
    return count;
}


int Shell_Sort(Vector<long>& v){
    int len=v.size();
    long long count=0;
    for (int gap=len/2;gap>0;gap/=2){
        for (int i=gap;i<len;++i) {
            int temp=v[i], j;
            for (j=i-gap;j>0&&v[j]>temp;j-=gap) v[j+gap]=v[j],++count;
            v[j+gap]=temp;++count;
        }
    }
    return count;
}
```

```
long long heapify(Vector<long>& v,int curNode, int size){
    long long count=0;
    int left=2*curNode+1;
    int right=2*curNode+2;
    int max=curNode;
    if (left<size) max=v[max]>v[left]?max:left;
    if (right<size) max=v[max]>v[right]?max:right;
    if (max!=curNode) {
        swap(v[max],v[curNode]);
        ++count;
        count+=heapify(v,max,size);
    }
    return count;
}

int Heap_Sort(Vector<long>& v){
    int len=v.size();
    long long count=0;
    for (int i = len - 1; i >= 0; i--)
        count += heapify(v, i, len);
    for (int size=len;size>0;size--) {
            count += heapify(v, 0, size);
        swap(v[0],v[size-1]);
    }
    return count;
}

int Msort(Vector<long>& v, int L, int R){
    long long count=0;
    if (L>=R) return 0;
    int mid=(L+R)/2;
    count+=Msort(v,L,mid);
    count+=Msort(v,mid+1,R);
    int p1=L, p2=mid+1;
    Vector<long> temp;
    while (p1<=mid && p2<=R) temp.push_back(v[p1]<v[p2]?v[p1++]:v[p2++]),++count;
    while (p1<=mid) temp.push_back(v[p1++]),++count;
    while (p2<=R) temp.push_back(v[p2++]),++count;
    for (int i=0;i<temp.size();++i)
        v[L+i]=temp[i];
    return count;
}

int Merge_Sort(Vector<long>& v){
    int len=v.size();
    long long count=0;
    count += Msort(v,0,len-1);
    return count;
```

```
}

int maxbit(Vector<long>& v) {
    int max=0, len=v.size();
    for (int i=0;i<len;++i) max=v[max]<v[i]?i:max;
    int d=0,t=v[max];
    while (t>=10) t/=10,++d;
    return d+1;
}

int Radix_Sort(Vector<long>& v){
    int len=v.size();
    long long count=0;
    int counter[10], radix=1;
    for (int i=0;i<maxbit(v);++i) {
        Vector<long> tmp(v.size());
        memset(counter,0, sizeof(counter));
        for (int j = 0; j < len; ++j) ++counter[(v[j] / radix) % 10];
        for (int j = 1; j < 10; ++j) counter[j] += counter[j - 1];
        for (int j = len - 1; j >= 0; --j) tmp[--counter[(v[j] / radix) % 10]] = v[j],
             ++count;
        v=tmp;
        radix*=10;
    }
    return count;
}
```

2.4 System design

The system first outputs a user description and a user prompt. After the user enters a random number to be generated, the corresponding sorting function is executed according to the operation code (code) input by the user. After sorting is completed, the sorting consumption time and the number of exchanges are output.

Chapter 3

Implementation

3.1 Bubble Sort Function

3.1.1 Bubble Sort core code

```
    int Bubble_Sort(Vector<long>& v){
    int len=v.size();
    long long count=0;
    for (int i=0;i<len-1;++i)
        for (int j=0;j<len-i-1;++j) {
            if (v[j] > v[j + 1]) swap(v[j], v[j + 1]);
            count++;
        }
    return count;
}
```

3.1.2 Bubble Sort flowchart

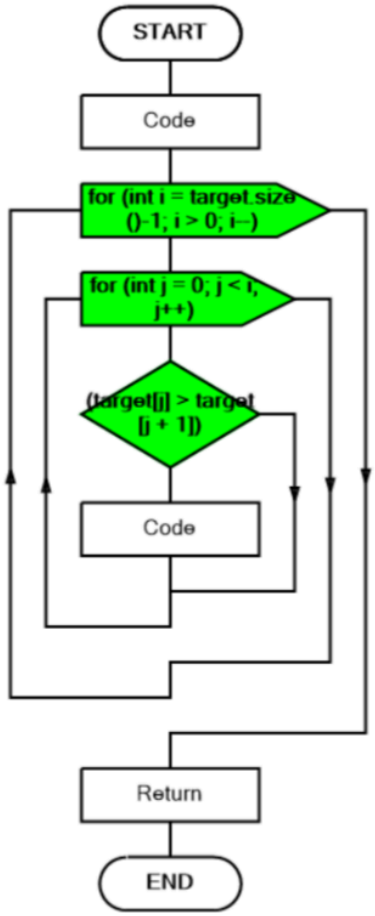

Figure 3.1: Bubble Sort flowchart

3.2 Selection Sort Function

3.2.1 Selection Sort code

```
    int Selection_Sort(Vector<long>& v){
    int len=v.size();
    long long count=0;
    for (int i=0;i<len;++i) {
        int p=0;
        for (int j = i+1; j < len; ++j) {
            count++;
            if (v[j] < v[p]) p = j;
        }
        swap(v[i],v[p]);
    }
    return count;
}
```

3.2.2 Selection Sort flowchart

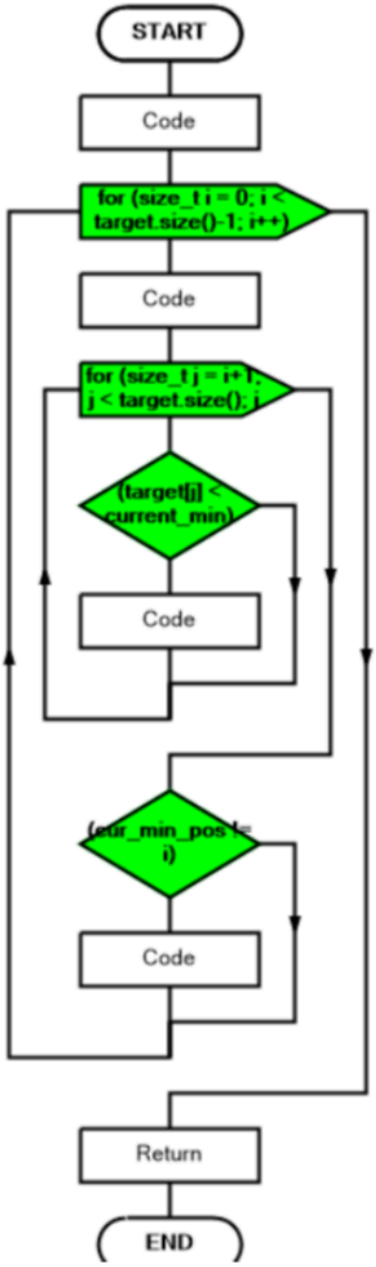

Figure 3.2: Selection Sort flowchart

3.3 Insertion Sort Function

3.3.1 Insertion Sort code

```
int Insertion_Sort(Vector<long>& v){
int len=v.size();
long long count=0;
for (int i=0;i<len;++i) {
```

```
        int temp=v[i];
        int j=i;
        while (j>=0) {
            count++;
            if (j>0&&v[j-1]>temp) v[j]=v[j-1];
            else {v[j]=temp; break;}
            j--;
        }
    }
    return count;
}
```

3.3.2 Insertion Sort flowchart

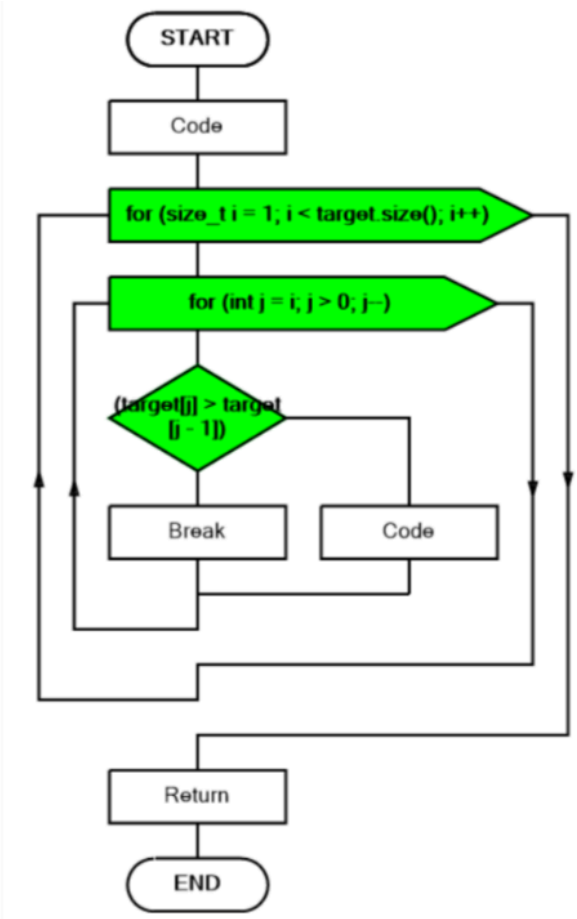

Figure 3.3: Insertion Sort flowchart

3.4 Shell Sort Function

3.4.1 Shell Sort code

```
int Shell_Sort(Vector<long>& v){
```

```
    int len=v.size();
    long long count=0;
    for (int gap=len/2;gap>0;gap/=2){
        for (int i=gap;i<len;++i) {
            int temp=v[i], j;
            for (j=i-gap;j>0&&v[j]>temp;j-=gap) v[j+gap]=v[j],++count;
            v[j+gap]=temp;++count;
        }
    }
    return count;
}
```

3.4.2 Shell Sort flowchart

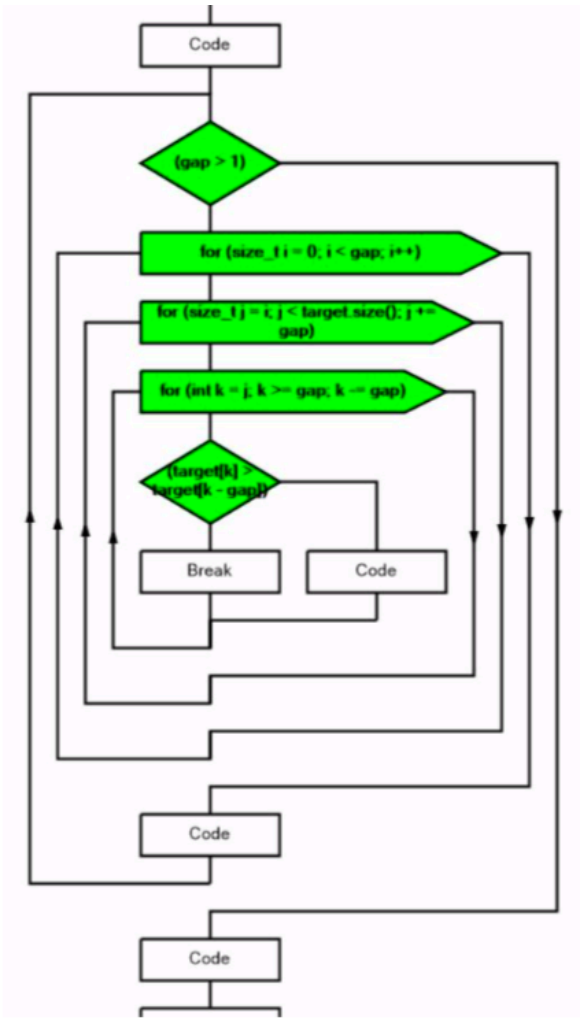

Figure 3.4: Shell Sort flowchart

3.5 Quick Sort Function

3.5.1 Quick Sort code

```
int qsort(Vector<long>& v, int L, int R){
    int i=L, j=R, x=v[L];
    long long count=0;
    if (L>=R) return 0;
    while (i<j) {
        while (i<j&&v[j]>=x) --j;
        if (i<j) v[i++]=v[j],++count;
        while (i<j&&v[i]<x) ++i;
        if (i<j) v[j--]=v[i],++count;
    }
    v[i]=x; ++count;
    count+=qsort(v,L,i-1);
    count+=qsort(v,i+1,R);
    return count;
}

int Quick_Sort(Vector<long>& v){
    int len=v.size();
    long long count=0;
    count+=qsort(v,0,len-1);
    return count;
}
```

3.5.2 Quick Sort flowchart

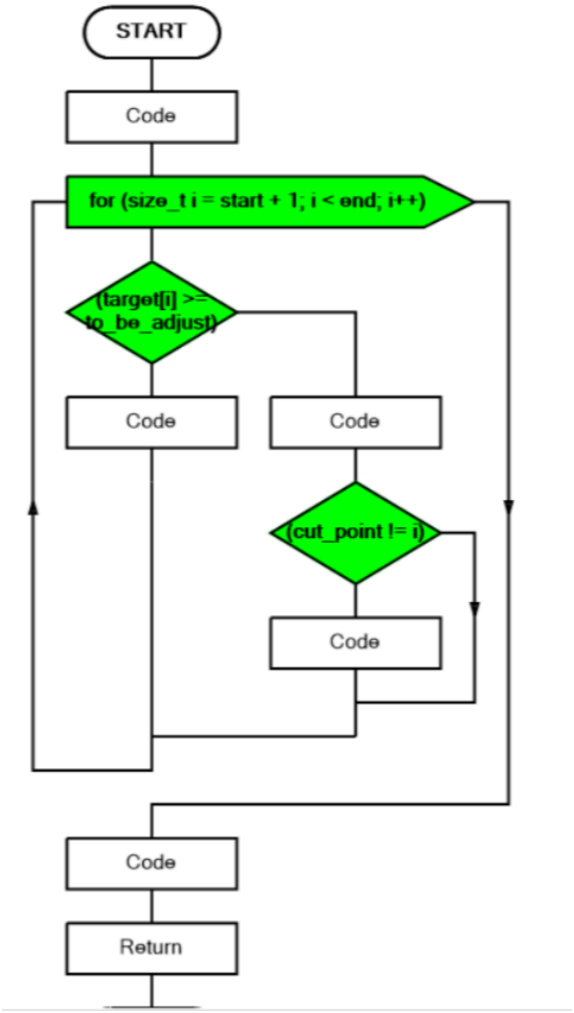

Figure 3.5: Quick Sort flowchart

3.6 Heap Sort Function

3.6.1 Heap Sort code

```
long long heapify(Vector<long>& v,int curNode, int size){
    long long count=0;
    int left=2*curNode+1;
    int right=2*curNode+2;
    int max=curNode;
    if (left<size) max=v[max]>v[left]?max:left;
    if (right<size) max=v[max]>v[right]?max:right;
    if (max!=curNode) {
        swap(v[max],v[curNode]);
        ++count;
```

```
        count+=heapify(v,max,size);
    }
    return count;
}

int Heap_Sort(Vector<long>& v){
    int len=v.size();
    long long count=0;
    for (int i = len - 1; i >= 0; i--)
        count += heapify(v, i, len);
    for (int size=len;size>0;size--) {
           count += heapify(v, 0, size);
        swap(v[0],v[size-1]);
    }
    return count;
}
```

3.6.2 Heap Sort flowchart

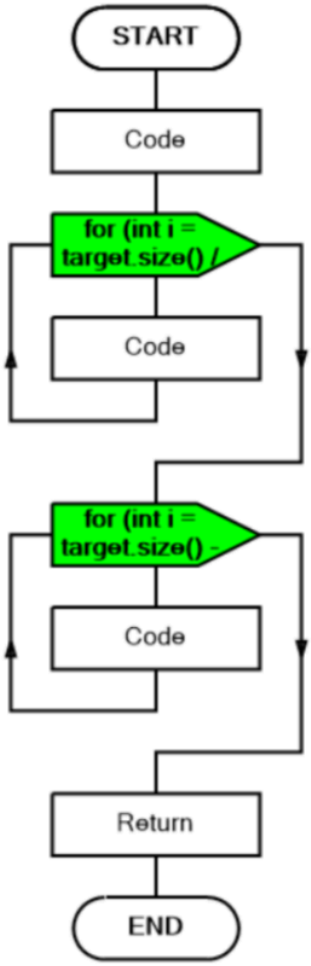

Figure 3.6: Heap Sort flowchart

3.7 Merge Sort Function

3.7.1 Merge Sort code

```
int Msort(Vector<long>& v, int L, int R){
    long long count=0;
    if (L>=R) return 0;
    int mid=(L+R)/2;
    count+=Msort(v,L,mid);
    count+=Msort(v,mid+1,R);
    int p1=L, p2=mid+1;
    Vector<long> temp;
    while (p1<=mid && p2<=R) temp.push_back(v[p1]<v[p2]?v[p1++]:v[p2++]),++count;
```

```
    while (p1<=mid) temp.push_back(v[p1++]),++count;
    while (p2<=R) temp.push_back(v[p2++]),++count;
    for (int i=0;i<temp.size();++i)
        v[L+i]=temp[i];
    return count;
}

int Merge_Sort(Vector<long>& v){
    int len=v.size();
    long long count=0;
    count += Msort(v,0,len-1);
    return count;
}
```

3.7.2 Merge Sort flowchart

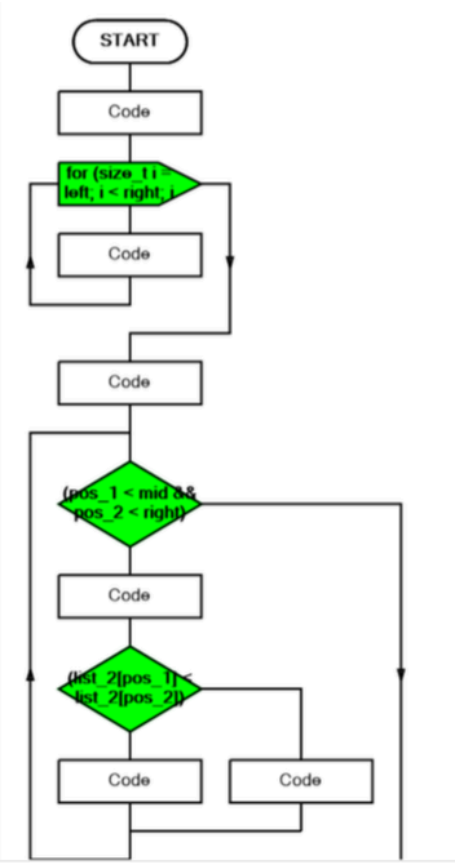

Figure 3.7: Merge Sort flowchart

3.8 Radix Sort Function

3.8.1 Radix Sort code

```
int maxbit(Vector<long>& v) {
    int max=0, len=v.size();
    for (int i=0;i<len;++i) max=v[max]<v[i]?i:max;
    int d=0,t=v[max];
    while (t>=10) t/=10,++d;
    return d+1;
}

int Radix_Sort(Vector<long>& v){
    int len=v.size();
    long long count=0;
    int counter[10], radix=1;
    for (int i=0;i<maxbit(v);++i) {
        Vector<long> tmp(v.size());
        memset(counter,0, sizeof(counter));
        for (int j = 0; j < len; ++j) ++counter[(v[j] / radix) % 10];
        for (int j = 1; j < 10; ++j) counter[j] += counter[j - 1];
        for (int j = len - 1; j >= 0; --j) tmp[--counter[(v[j] / radix) % 10]] = v[
            j], ++count;
        v=tmp;
        radix*=10;
    }
    return count;
}
```

3.8.2 Radix Sort flowchart

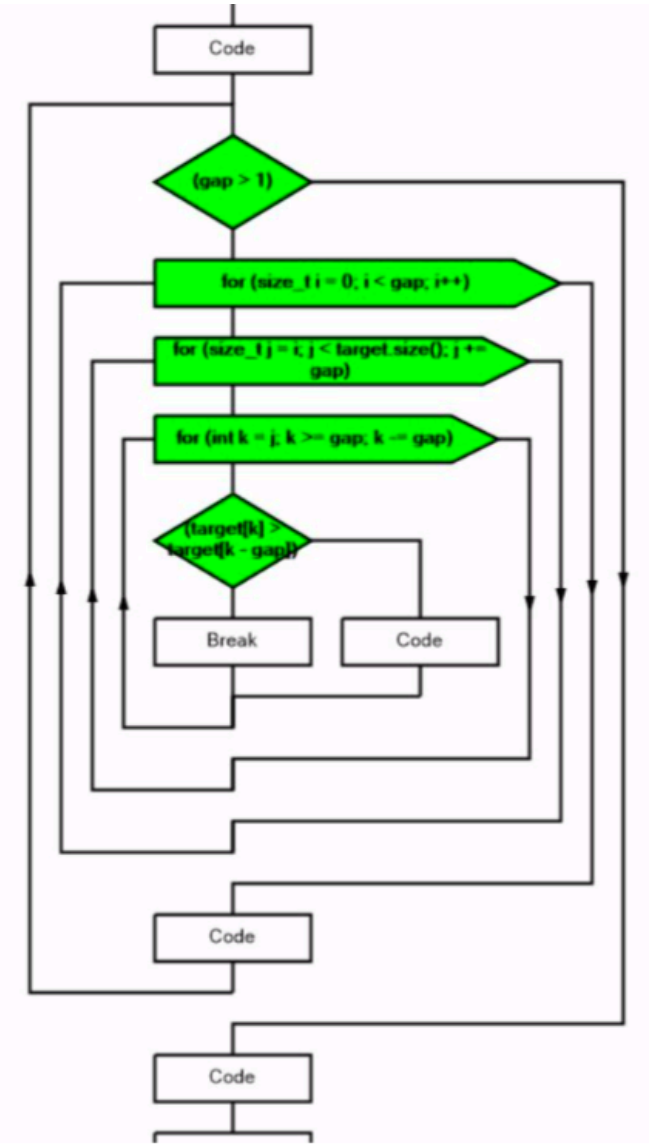

Figure 3.8: Radix Sort flowchart

Chapter 4

Test

4.1 Function Tests

4.1.1 n=100

Test Case:

n=100

Program Result:

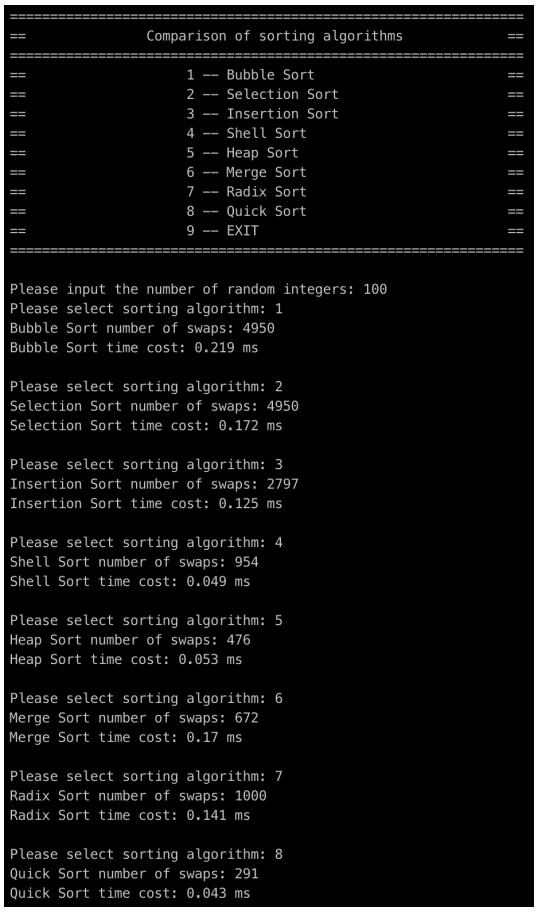

Figure 4.1: Screenshots

4.1.2 n=1000

Test Case:

n=1000

Program Result:

```
============================================================
==              Comparison of sorting algorithms          ==
============================================================
==                   1 -- Bubble Sort                     ==
==                   2 -- Selection Sort                  ==
==                   3 -- Insertion Sort                  ==
==                   4 -- Shell Sort                      ==
==                   5 -- Heap Sort                       ==
==                   6 -- Merge Sort                      ==
==                   7 -- Radix Sort                      ==
==                   8 -- Quick Sort                      ==
==                   9 -- EXIT                            ==
============================================================

Please input the number of random integers: 1000
Please select sorting algorithm: 1
Bubble Sort number of swaps: 499500
Bubble Sort time cost: 8.352 ms

Please select sorting algorithm: 2
Selection Sort number of swaps: 499500
Selection Sort time cost: 7.141 ms

Please select sorting algorithm: 3
Insertion Sort number of swaps: 248083
Insertion Sort time cost: 3.338 ms

Please select sorting algorithm: 4
Shell Sort number of swaps: 15901
Shell Sort time cost: 0.605 ms

Please select sorting algorithm: 5
Heap Sort number of swaps: 8096
Heap Sort time cost: 0.797 ms

Please select sorting algorithm: 6
Merge Sort number of swaps: 9976
Merge Sort time cost: 0.945 ms

Please select sorting algorithm: 7
Radix Sort number of swaps: 10000
Radix Sort time cost: 1.566 ms

Please select sorting algorithm: 8
Quick Sort number of swaps: 4397
Quick Sort time cost: 0.407 ms
```

Figure 4.2: Screenshots

4.1.3 n=10000

Test Case:

n=10000

Program Result:

```
============================================================
==               Comparison of sorting algorithms           ==
============================================================
==                    1 -- Bubble Sort                      ==
==                    2 -- Selection Sort                   ==
==                    3 -- Insertion Sort                   ==
==                    4 -- Shell Sort                       ==
==                    5 -- Heap Sort                        ==
==                    6 -- Merge Sort                       ==
==                    7 -- Radix Sort                       ==
==                    8 -- Quick Sort                       ==
==                    9 -- EXIT                             ==
============================================================

Please input the number of random integers: 10000
Please select sorting algorithm: 1
Bubble Sort number of swaps: 49995000
Bubble Sort time cost: 554.451 ms

Please select sorting algorithm: 2
Selection Sort number of swaps: 49995000
Selection Sort time cost: 265.588 ms

Please select sorting algorithm: 3
Insertion Sort number of swaps: 25246611
Insertion Sort time cost: 195.363 ms

Please select sorting algorithm: 4
Shell Sort number of swaps: 263982
Shell Sort time cost: 8.567 ms

Please select sorting algorithm: 5
Heap Sort number of swaps: 114162
Heap Sort time cost: 6.299 ms

Please select sorting algorithm: 6
Merge Sort number of swaps: 133616
Merge Sort time cost: 11.325 ms

Please select sorting algorithm: 7
Radix Sort number of swaps: 100000
Radix Sort time cost: 11.125 ms

Please select sorting algorithm: 8
Quick Sort number of swaps: 59692
Quick Sort time cost: 4.884 ms
```

Figure 4.3: Screenshots

4.1.4 n=100000

Test Case:

n=100000

Program Result:

```
============================================================
==               Comparison of sorting algorithms         ==
============================================================
==                    1 -- Bubble Sort                    ==
==                    2 -- Selection Sort                 ==
==                    3 -- Insertion Sort                 ==
==                    4 -- Shell Sort                     ==
==                    5 -- Heap Sort                      ==
==                    6 -- Merge Sort                     ==
==                    7 -- Radix Sort                     ==
==                    8 -- Quick Sort                     ==
==                    9 -- EXIT                           ==
============================================================

Please input the number of random integers: 100000
Please select sorting algorithm: 1
Bubble Sort number of swaps: 704982704
Bubble Sort time cost: 54353.4 ms

Please select sorting algorithm: 2
Selection Sort number of swaps: 704982704
Selection Sort time cost: 24445.2 ms

Please select sorting algorithm: 3
Insertion Sort number of swaps:  1790390294
Insertion Sort time cost: 16949.9 ms

Please select sorting algorithm: 4
Shell Sort number of swaps: 4452360
Shell Sort time cost: 59.335 ms

Please select sorting algorithm: 5
Heap Sort number of swaps: 1474747
Heap Sort time cost: 54.305 ms

Please select sorting algorithm: 6
Merge Sort number of swaps: 1668928
Merge Sort time cost: 72.042 ms

Please select sorting algorithm: 7
Radix Sort number of swaps: 1000000
Radix Sort time cost: 61.112 ms

Please select sorting algorithm: 8
Quick Sort number of swaps: 749759
Quick Sort time cost: 27.571 ms
```

Figure 4.4: Screenshots

Appendix A

Frequently Asked Questions

A.1 Why this document looks so sparse?

This document is built using LaTeX and is arranged in book format. There may be blank pages before and after each chapter.

A.2 Why is there header files in this project?

I wrote basic stl header files, such as vector.h, etc. Except for the header files I wrote, the only header files used in all projects are iostream and string.h