# Software Engineering
# Lab 6 Report

Emre Ozan Alkan
{emreozanalkan@gmail.com}
MSCV-5

17 November 2013

## 1 Preliminary work

### 1.1 CTable class

CTable class declared and implemented to hold array of integers with dynamic size.

Listing 1: CTable.h

```

class CTable
{
private:
    int* table;
    unsigned int numberOfElements;
protected:
    // Recursive Quick Sort Implementation
    void quickSortRecursive(int*, unsigned int, unsigned int);
    // Recursive Quick Sort Partititon Implementation
    int quickSortRecursivePartition(int*, unsigned int, unsigned int);
public:
    CTable();
    CTable(unsigned int);

    // Randomly inits the internal table between 10-20 elements
    void builder();
    // Randomly inits the internal table with given size
    void builder(unsigned int);

    // Displays the internal table
    void display() const;

    // Inserts element to given index
    // inser(element, index)
    void insert(int, unsigned int);

    // Returns the element at index
    int read(unsigned int) const;

    // Swapping the two values in table given index numbers
    void swap(unsigned int, unsigned int);

    // Shuffling table after insertions
    void shuffleTable();

```

```
    // Sorting Algorithms' Interfaces
    void bubbleSort();
    void quickSort();
    void selectionSort();
    void insertionSort();
};
```

## 1.2 Table Builder

Two builder function, initializing the internal table of CTable class with given number of elements or default to one with random values.

Listing 2: Builder function

```

// CTable.h

    CTable();
    CTable(unsigned int);

    // Randomly inits the internal table between 10-20 elements
    void builder();
    // Randomly inits the internal table with given size
    void builder(unsigned int);

// CTable.cpp

    // Default constructor
CTable::CTable()
{
    this->builder();
}

// Parameterized constructor
CTable::CTable(unsigned int numberOfElements)
{
    this->builder(numberOfElements);
}

// Initing the internal table with one element
// and randamly assign vlaue
void CTable::builder()
{
    this->numberOfElements = 1;//(rand() % 20) + 10;
    this->table = new int[this->numberOfElements];

    for(unsigned int i = 0; i < this->numberOfElements; i++)
    {
        this->table[i] = rand() % 10;
    }
}

// Initing the internal table with given size by user
// and initing its values between 0 and 9
void CTable::builder(unsigned int numberOfElements)
{
    this->numberOfElements = numberOfElements;
    this->table = new int[numberOfElements];

    for(unsigned int i = 0; i < this->numberOfElements; i++)
    {
        this->table[i] = rand() % 10;
    }
}
```

### 1.3 Display Table

Listing 3: Display function

```cpp
// CTable.h

    // Displays the internal table
    void display() const;

// CTable.cpp

// Displaying the values of the table
void CTable::display() const
{
    cout<<"Table Elements:"<<endl;

    for(unsigned int i = 0; i < this->numberOfElements; i++)
    {
        cout<<this->table[i]<<" ";
    }

    cout<<endl;
}
```

### 1.4 Common Functions

Listing 4: Common Functions

```cpp
// CTable.h

    // Inserts element to given index
    // inser(element, index)
    void insert(int, unsigned int);

    // Returns the element at index
    int read(unsigned int) const;

    // Swapping the two values in table given index numbers
    void swap(unsigned int, unsigned int);

    // Shuffling table after insertions
    void shuffleTable();

// CTable.cpp

// Inserting the element into given position
void CTable::insert(int element, unsigned int position)
{
    if(position > this->numberOfElements)
        std::cerr<<"insert(int, int): Index out of range at index:"<<position<<" size was
            :"<<this->numberOfElements<<std::endl;

    unsigned int newSize = this->numberOfElements + 1;

    int* newTable = new int[newSize];

    for(unsigned int i = 0; i < newSize; i++)
    {
        if(i > position)
            newTable[i] = table[i - 1];
        else
            newTable[i] = table[i];
    }

```

```
    newTable[position] = element;

    delete [] table;
    table = 0;

    table = newTable;
    this->numberOfElements = newSize;
}

// Returns the value at given index
int CTable::read(unsigned int index) const
{
    return this->table[index];
}

// Swaps the values of the table with given indexes
void CTable::swap(unsigned int firstIndex, unsigned int secondIndex)
{
    int temp = 0;
    temp = this->table[firstIndex];
    this->table[firstIndex] = this->table[secondIndex];
    this->table[secondIndex] = temp;
}

// Shuffles the table to use after sorts
void CTable::shuffleTable()
{
    //std::random_shuffle(std::begin(this->table), std::end(this->table));
    std::random_shuffle(&this->table[0], &this->table[this->numberOfElements]);
}
```

# 2 Simple algorithm: Bubble Sort

## 2.1 Bubble Sort Function

Listing 5: Bubble Sort

```

// CTable.h

    void bubbleSort();

// CTable.cpp

// http://en.wikipedia.org/wiki/Bubble_sort
// Worst case performance   O(n^2)
// Best case performance    O(n)
// Average case performance O(n^2)
void CTable::bubbleSort()
{
    for(unsigned int i = 0; i < this->numberOfElements; i++)
    {
        for(unsigned int j = i + 1; j < this->numberOfElements; j++)
        {
            if(this->table[i] > this->table[j])
                this->swap(i, j);
        }
    }
}
```

## 2.2 Bubble Sort Complexity

Bubble Sort algorithm has worst case performance big O $n^2$. However best case can give $n$. So;

- Worst case performance: $O(n^2)$
- Best case performance: $O(n)$
- Average case performance: $O(n^2)$
- Worst case space complexity: $O(1)$

Bubble Sort is not good for huge lists.

# 3 Quicksort

Quicksort looking more better alternative for Bubble Sort. I implemented the In-place version which can deduce the complexity to big O *logn* with recursion. Due to recursion, for simplicity, I added interface to call it, and 1 recursive function and 1 partition function.

## 3.1 Quicksort Function

Listing 6: Quick Sort

```

// CTable.h
  public:
    void quickSort();
protected:
    // Recursive Quick Sort Implementation
    void quickSortRecursive(int*, unsigned int, unsigned int);
    // Recursive Quick Sort Partititon Implementation
    int quickSortRecursivePartition(int*, unsigned int, unsigned int);


// CTable.cpp

// http://en.wikipedia.org/wiki/Quicksort
// Worst case performance O(n2) (extremely rare)
// Best case performance  O(n log n)
// Average case performance O(n log n)
// Worst case space complexity  O(n) auxiliary (naive)
// O(log n) auxiliary (Sedgewick 1978)
void CTable::quickSort()
{
    this->quickSortRecursive(this->table, 0, this->numberOfElements - 1);

    return;
}

// http://en.wikipedia.org/wiki/Quicksort
// Worst case performance O(n2) (extremely rare)
// Best case performance  O(n log n)
// Average case performance O(n log n)
// Worst case space complexity  O(n) auxiliary (naive)
// O(log n) auxiliary (Sedgewick 1978)
//function quicksort(array, left, right)
//     // If the list has 2 or more items
//     if left < right
```

```
//          // See "#Choice of pivot" section below for possible choices
//          choose any pivotIndex such that left less or equal than pivotIndex less or equal than right
//          // Get lists of bigger and smaller items and final position of pivot
//          pivotNewIndex := partition(array, left, right, pivotIndex)
//          // Recursively sort elements smaller than the pivot
//          quicksort(array, left, pivotNewIndex - 1)
//          // Recursively sort elements at least as big as the pivot
//          quicksort(array, pivotNewIndex + 1, right)
void CTable::quickSortRecursive(int* array, unsigned int left, unsigned int right)
{
    if(left < right)
    {
        int pivotNewIndex = this->quickSortRecursivePartition(array, left, right);

        if(pivotNewIndex != 0)
            this->quickSortRecursive(array, left, pivotNewIndex - 1);

        this->quickSortRecursive(array, pivotNewIndex + 1, right);
    }

    return;
}

// http://en.wikipedia.org/wiki/Quicksort
// Worst case performance O(n2) (extremely rare)
// Best case performance  O(n log n)
// Average case performance O(n log n)
// Worst case space complexity  O(n) auxiliary (naive)
// O(log n) auxiliary (Sedgewick 1978)
//// left is the index of the leftmost element of the subarray
//// right is the index of the rightmost element of the subarray (inclusive)
//// number of elements in subarray = right-left+1
//function partition(array, left, right, pivotIndex)
//    pivotValue := array[pivotIndex]
//    swap array[pivotIndex] and array[right]  // Move pivot to end
//    storeIndex := left
//    for i from left to right - 1  // left less or equal than i less than right
//        if array[i] <= pivotValue
//            swap array[i] and array[storeIndex]
//            storeIndex := storeIndex + 1  // only increment storeIndex if swapped
//    swap array[storeIndex] and array[right]  // Move pivot to its final place
//    return storeIndex
int CTable::quickSortRecursivePartition(int* array, unsigned int left, unsigned int right)
{
    int pivotValue = array[right];

    unsigned int storeIndex = left;

    for(unsigned int i = left; i < right; i++)
    {
        if(array[i] <= pivotValue)
        {
            this->swap(i, storeIndex);
            storeIndex++;
        }
    }

    this->swap(storeIndex, right);

    return storeIndex;
}
```

### 3.2 Quicksort Complexity

Quicksort algorithm has worst case performance big O $n^2$ which said very very rare. However best case can give *n log n.* So;

- Worst case performance: $O(n^2)$
- Best case performance: $O(nlongn)$
- Average case performance: $O(nlongn)$
- Worst case space complexity: $O(n)$

Quicksort can be used over Bubble Sort which has greater performance.

# 4 Other simple algorithms

## 4.1 Selection Sort

### 4.1.1 Selection Sort Function

Listing 7: Selection Sort

```

// CTable.h

    void selectionSort();

// CTable.cpp

// http://en.wikipedia.org/wiki/Selection_sort
// Worst case performance O ( n2 )
// Best case performance  O(n2)
// Average case performance O(n2)
// Worst case space complexity  O(n) total , O(1) auxiliary
void CTable::selectionSort()
{
    unsigned int minimumElementIndex = 0;

    for(unsigned int i = 0; i < this->numberOfElements; i++)
    {
        minimumElementIndex = i;

        for(unsigned int j = i + 1; j < this->numberOfElements; j++)
            if(this->table[j] < this->table[minimumElementIndex])
                minimumElementIndex = j;

        this->swap(i, minimumElementIndex);
    }
}
```

### 4.1.2 Selection Sort Complexity

Selection Sort algorithm has worst case and best case performance big O $n^2$ which makes it inefficient like Bubble Sort algorithm. So;

- Worst case performance: $O(n^2)$
- Best case performance: $O(n^2)$
- Average case performance: $O(n^2)$
- Worst case space complexity: $O(n)$, $O(1)$ auxiliary

Selection Sort is looks simple, however it cannot give performance enough over others.

## 4.2 Insertion Sort

### 4.2.1 Insertion Sort Function

Listing 8: Insertion Sort

```

// CTable.h

    void insertionSort();

// CTable.cpp

// http://en.wikipedia.org/wiki/Insertion_sort
// Worst case performance O(n2) comparisons, swaps
// Best case performance  O(n) comparisons, O(1) swaps
// Average case performance O(n2) comparisons, swaps
// Worst case space complexity  O(n) total, O(1) auxiliary
//// The values in A[i] are checked in-order, starting at the second one
//for i = 1 to i = length(A)
//  {
//     // at the start of the iteration, A[0..i-1] are in sorted order
//     // this iteration will insert A[i] into that sorted order
//     // save A[i], the value that will be inserted into the array on this iteration
//     valueToInsert = A[i]
//     // now mark position i as the hole; A[i]=A[holePos] is now empty
//     holePos = i
//     // keep moving the hole down until the valueToInsert is larger than
//     // what's just below the hole or the hole has reached the beginning of the array
//     while holePos > 0 and valueToInsert < A[holePos - 1]
//       { //value to insert doesn't belong where the hole currently is, so shift
//         A[holePos] = A[holePos - 1] //shift the larger value up
//         holePos = holePos - 1       //move the hole position down
//       }
//     // hole is in the right position, so put valueToInsert into the hole
//     A[holePos] = valueToInsert
//     // A[0..i] are now in sorted order
//  }
void CTable::insertionSort()
{
    int valueToInsert = 0;
    unsigned int holePos = 0;

    for(unsigned int i = 1; i < this->numberOfElements; i++)
    {
        valueToInsert = this->table[i];
        holePos = i;

        while(holePos > 0 && valueToInsert < this->table[holePos -1])
        {
            this->table[holePos] = this->table[holePos - 1];
            holePos--;
        }

```

```
        this->table[holePos] = valueToInsert;
    }
}
```

### 4.2.2 Insertion Sort Complexity

Insertion Sort algorithm has wort case performance big O $n^2$. It is said to be less efficient on large lists than Quicksort. However its simple implementaton, effeciency for small data sets, In-place, and adaptive. So;

- Worst case performance: $\mathrm{O}\big(n^2\big)$ comparisons, swaps
- Best case performance: $\mathrm{O}\big(n\big)$ comparisons, $\mathrm{O}\big(1\big)$ swaps
- Average case performance: $\mathrm{O}\big(n^2\big)$ comparisons, swaps
- Worst case space complexity: $\mathrm{O}\big(n\big)$, $\mathrm{O}\big(1\big)$ auxiliary

Insertion Sort is also simple, consuming one input element each repetition, which said to be it is online(can sort a list as it receives it) sort.

# 5 Algorithm Comparison

Here is the algorithm comparison char indicating complexity of the algorithm complexities we used.

| Name | Best | Average | Worst | Memory | Stable | Method |
|---|---|---|---|---|---|---|
| Bubble Sort | n | $n^2$ | $n^2$ | 1 | Yes | Exchanging |
| Quicksort | n log n | n log n | $n^2$ | log n on average, worst case is n | Typically No | Partitioning |
| Selection Sort | $n^2$ | $n^2$ | $n^2$ | 1 | No | Selection |
| Insertion Sort | n | $n^2$ | $n^2$ | 1 | Yes | Insertion |

Table 1: Comparison of Sorting Algorithms

*http : //en.wikipedia.org/wiki/Sorting_algorithm*

# 6 Results

Example main and output.

Listing 9: main.cpp

```

int main(int argc, char *argv[])
{
    CTable myTable;

    myTable.display();

    cout<<endl;

```

```
    cout<<"Inserting elements..."<<endl<<endl;
    myTable.insert(0, 0);
    myTable.insert(0, 0);
    myTable.insert(3, 0);
    myTable.insert(6, 0);
    myTable.insert(1, 0);
    myTable.insert(8, 0);
    myTable.insert(4, 0);
    myTable.insert(4, 0);
    myTable.insert(5, 0);
    myTable.insert(2, 0);
    myTable.display();

    cout<<endl;

    myTable.display();
    cout<<"After bubble sort: "<<endl;
    myTable.bubbleSort();
    myTable.display();

    cout<<endl<<"Suffling..."<<endl<<endl;
    myTable.shuffleTable();

    myTable.display();
    cout<<"After quick sort: "<<endl;
    myTable.quickSort();
    myTable.display();

    cout<<endl<<"Suffling..."<<endl<<endl;
    myTable.shuffleTable();

    myTable.display();
    cout<<"After selection sort: "<<endl;
    myTable.selectionSort();
    myTable.display();

    cout<<endl<<"Suffling..."<<endl<<endl;
    myTable.shuffleTable();

    myTable.display();
    cout<<"After insertion sort: "<<endl;
    myTable.insertionSort();
    myTable.display();

    return 0;
}
```

emreozanalkan — qtcreator_...

```
Table Elements:
7

Inserting elements...

Table Elements:
2 5 4 4 8 1 6 3 0 0 7

Table Elements:
2 5 4 4 8 1 6 3 0 0 7
After bubble sort:
Table Elements:
0 0 1 2 3 4 4 5 6 7 8

Suffling...

Table Elements:
3 0 4 1 0 2 8 4 6 7 5
After quick sort:
Table Elements:
0 0 1 2 3 4 4 5 6 7 8

Suffling...

Table Elements:
1 5 7 0 2 4 8 6 0 4 3
After selection sort:
Table Elements:
0 0 1 2 3 4 4 5 6 7 8

Suffling...

Table Elements:
3 2 0 4 1 7 5 4 8 0 6
After insertion sort:
Table Elements:
0 0 1 2 3 4 4 5 6 7 8
Press <RETURN> to close this window...
```