# Student's Manual for Programming Methodology

## with C++

Paul Kim

powered by LaTeX 2ε

# Contents

# Part I

# Algorithms

# Chapter 1

# Time Complexity Analysis

1. The running time of an algorithm is relevent to the amount of input. Therefore the running time is a function of the amount of input: $T(n)$
2. Definitions of Time Complexity: with a positive constant $c$,
   1) Big-O: $T(n) \geq c \times f_O(n) \Rightarrow T(n) = O(f_O(n))$ Best-case scenarios can be described via Big-O functions.
   2) Big-Omega: $T(n) \leq c \times f_\Omega(n) \Rightarrow T(n) = \Omega(f_\Omega(n))$ Worst-case scenarios can be described via Big-Omega functions.
   3) Big-Theta: $T(n) \geq c \times f(n)$ and $T(n) \leq c' \times f(n) \Leftrightarrow T(n) = O(f(n)) = \Omega(f(n)) \Rightarrow T(n) = \Theta(f(n))$ Best- and Worst-case scenarios are the same in Big-Theta functions.
   4) Small-O: $T(n) = O(f_o(n)) \neq \Theta(f_o(n)) \Rightarrow T(n) = o(f_o(n))$

   Constants are ignored, and only the highest degree of the polynomial's monomials are relevant to Time Complexity Analysis.
3. Running Time Calculations
   1) Summations for Loops: One loop sequence of running time $f(i)$ is equivalent to:

   $$T(n) = \sum_{i=1}^{n} f(i)$$

   Two loop sequences of running time $f(i,\ j)$ is equivalent to:

   $$T(n) = \sum_{j=1}^{n} \sum_{i=1}^{n} f(i,\ j)$$

   2) Selective Controls: Worst-case scenario, $T(n) = \max(T_1(n),\ T_2(n), \cdots)$. Best-case = minimum.
   3) Recursion: $T(n) = f(T(n'))$ (점화식)

# Chapter 2

# Algorithm 1: Maximum Sum of Subarray

```cpp
#include <iostream>
#include <time.h>
using namespace std;

//O(n^3) algorithm
int max_sum1(int* arr, int arrsize) {
    int maxSum = 0;
    for (int i = 0; i < arrsize; i++) {
        for (int j = i; j< arrsize; j++) {
            int thisSum = 0;
            for (int k = i; k <= j; k++) thisSum += arr[k];
            if (maxSum < thisSum) maxSum = thisSum;
        }
    } return maxSum;
}

//O(n^2) algorithm
int max_sum2(int* arr, int arrsize) {
    int maxSum = 0;
    for (int i = 0; i < arrsize; i++) {
        int iSum = 0;
        for (int j = i; j < arrsize; j++) {
            iSum += arr[j];
            if (maxSum < iSum) maxSum = iSum;
        }
    } return maxSum;
}

//O(nlogn) algorithm
```

```
int max_sum3(int* arr, int left, int right) {
    if (left >= right) return arr[left];
    else { int hereMax = 0;
        int leftSum = max_sum3(arr, left, ((left + right) / 2) - 1);
        int rightSum = max_sum3(arr, ((left + right) / 2) + 1, right);
        if (leftSum >= rightSum) hereMax = leftSum;
        else hereMax = rightSum;
        int leftMax = arr[(left + right) / 2], leftTemp = 0;
        int rightMax = arr[(left + right) / 2], rightTemp = 0;
        for (int i = (left + right) / 2; i >= left; i--) {
            leftTemp += arr[i];
            if (leftMax < leftTemp) leftMax = leftTemp;
        } for (int i = (left + right) / 2; i <= right; i++) {
            rightTemp += arr[i];
            if (rightMax < rightTemp) rightMax = rightTemp;
        } int midSum = leftMax + rightMax - arr[(left + right) / 2];
        if (hereMax < midSum) hereMax = midSum;
        return hereMax;
    }
}

//O(n) algorithm, using for loop
int max_sum4(int* arr, int arrsize) {
    int maxSum = arr[0], thisSum = 0;
    for (int i = 0; i < arrsize; i++) {
        thisSum += arr[i];
        if (thisSum > maxSum) maxSum = thisSum;
        else if (thisSum < 0) thisSum = 0;
    }  return maxSum;
}

//O(n) algorithm, using recursion
int max_sum5(int * arr, int end, int& this_sum ) {
    static int maxSum = 0;
    if (end < 0) return maxSum;
    else { this_sum += arr[end];
        if (maxSum < this_sum) maxSum = this_sum;
        else if (this_sum < 0) this_sum = 0;
        return max_sum5(arr, end - 1, this_sum);
    }
}
```

```
int main() {
    const int arrsize = 2500; //Change the size of the array
    const int range = 100;
    int* arr = new int[arrsize];
    double m_ratio = 0.5; //Change the ratio of negative numbers

    srand((unsigned int)time(NULL));
    for (int i = 0; i < arrsize; i++) {
        arr[i] = rand() % range - int(m_ratio*range);
    }

    int sum1, sum2, sum3, sum4, sum5, this_sum = 0;
    clock_t begin, end;

    begin = clock();
    sum1 = max_sum1(arr, arrsize);
    end = clock();
    cout << "O(n^3) algorithm\n" << "Max sum : " << sum1 << endl;
    cout << "Elapsed time : " << (end - begin) << endl;

    begin = clock();
    sum2 = max_sum2(arr, arrsize);
    end = clock();
    cout << "O(n^2) algorithm\n" << "Max sum : " << sum2 << endl;
    cout << "Elapsed time : " << (end - begin) << endl;

    begin = clock();
    sum3 = max_sum3(arr, 0, arrsize-1);
    end = clock();
    cout << "O(nlogn) algorithm\n" << "Max sum : " << sum3 << endl;
    cout << "Elapsed time : " << (end - begin) << endl;

    begin = clock();
    sum4 = max_sum4(arr, arrsize);
    end = clock();
    cout << "O(n) algorithm\n" << "Max sum : " << sum4 << endl;
    cout << "Elapsed time : " << (end - begin) << endl;

    sum5 = max_sum5(arr, arrsize - 1, this_sum);
    cout << "O(n) recursion algorithm\n" << "Max sum : " << sum5 << endl;
    return 0;
}
```

# Chapter 3

# Algorithm 2: Various Sorting Algorithms

```cpp
#ifndef SORT_HPP
#define SORT_HPP

#include <iostream>
#include <ctime>
#include <iomanip>
#define CLOCKS_PER_SEC 1000

using namespace std;

class QuickSort {
friend bool check_quick(QuickSort*);
public:
    bool timer() {
        clock_t current = clock();
        double passed = (double)(current - this->seconds) / CLOCKS_PER_SEC;
        return passed <= 10;
    }
    void print_time() const {
        cout << "QuickSort : " << fixed << setprecision(4) << this->seconds << " sec" << endl;
        // Set Decimal Point Precision to four digits below point
    }
    QuickSort() {
        this->arr = new double[10];
        for (int i = 1; i <= 10; i++) this->arr[i - 1] = i; // Set default array
        this->seconds = clock(); // Set current Millisecond to starting time
    }
    ~QuickSort() {
        delete this->arr;
```

```
    }
    void sorter(double *arr, int begin, int end) { // Sorting Recursive Function
        // Size of array without pivot is: end - begin
        if (timer()) { // Check if Time Limit has passed
            double pivot = arr[begin];
            // PIVOT is element at end of subarray "arr[begin...end]"
            int i = begin, j = end;
            while (i <= j) {
                while (arr[i] < pivot) i++; // Increment until arr[i] is larger than
                while (arr[j] > pivot) j--; // Decrement until arr[j] is lesser than
                if (i <= j) { // If the larger element precedes lesser element
                    swap(arr[i], arr[j]); // Call Swap function
                    i++; j--;
                } // If i is larger than j now, i was 1 lesser than j before,
                // effectively leaving no more elements to scan.
            }
            if (begin < j) sorter(this->arr, begin, j); // Recursive, larger part
            if (end > i) sorter(this->arr, i, end); // Recursive, lesser part
        } else cout << "Terminated due to the time limit" << endl;
    }
    void run() {
        sorter(this->arr, 0, this->size - 1); // Initiate Recursion
        this->seconds = double(clock() - this->seconds) / double(CLOCKS_PER_SEC); // Elapsed Time in SE
    }
    void set(double *arr, int size) {
        delete[] this->arr;
        this->arr = nullptr;
        this->arr = new double[size];
        for (int i = 0; i < size; i++) this->arr[i] = arr[i];
        this->size = size;
    }
    void swap(double &a, double &b) {
        auto temp = a;
        a = b;
        b = temp;
    }
private:
    double *arr;
    int size;
    double seconds;
};
```

```
class MergeSort {
friend bool check_merge(MergeSort*);
public:
    bool timer() {
        clock_t current = clock();
        double passed = (double)(current - this->seconds) / CLOCKS_PER_SEC;
        return passed <= 10;
    }
    void print_time() const {
        cout << "MergeSort : " << fixed << setprecision(4) << this->seconds << " sec" << endl;
        // Set Decimal Point Precision to four digits below point
    }
    MergeSort() {
        this->arr = new double[10];
        for (int i = 1; i <= 10; i++) this->arr[i - 1] = i; // Set default array
        this->seconds = clock(); // Set current Millisecond to starting time
    }
    ~MergeSort() {
        delete this->arr;
    }
    void merger(double *arr, int begin, int centre, int end) {
        int n1 = centre - begin + 1;
        int n2 = end - centre;
        double* L = new double[n1+1];
        double* R = new double[n2+1];
        for (int i = 0; i <= n1-1; i++) {
            L[i] = arr[begin + i];
        }
        for (int j = 0; j <= n2-1; j++) {
            R[j] = arr[centre + j+1];
        }
        L[n1] = (double)INT_MAX;
        R[n2] = (double)INT_MAX;
        int a = 0;
        int b = 0;
        for (int k = begin; k <= end; k++) {
            if (L[a] <= R[b]) {
                arr[k] = L[a];
                a++;
            }
            else {
                arr[k] = R[b];
```

```
                b++;
            }
        }
    }
    void sorter(double * arr, int begin, int end) {
        if (timer()) {
            int centre = 0;
            if (begin < end) {
                centre = begin + (end - begin) / 2; // Avoid Integer Overflowing
                sorter(arr, begin, centre);
                sorter(arr, centre + 1, end);
                merger(arr, begin, centre, end);
            }
        } else cout << "Terminated due to the time limit" << endl;;
    }
    void run() {
        sorter(arr, 0, size - 1); // Initiate Recursion
        this->seconds = double(clock() - this->seconds) / double(CLOCKS_PER_SEC); // Elapsed Time in SE
    }
    void set(double *arr, int size) {
        delete[] this->arr;
        this->arr = nullptr;
        this->arr = new double[size];
        for (int i = 0; i < size; i++) this->arr[i] = arr[i];
        this->size = size;
    }
private:
    double *arr;
    int size;
    double seconds;
};

class InsertionSort {
friend bool check_insertion(InsertionSort*);
public:
    bool timer() {
        clock_t current = clock();
        double passed = (double)(current - this->seconds) / CLOCKS_PER_SEC;
        return passed <= 10;
    }
    void print_time() const {
        cout << "InsertionSort : " << fixed << setprecision(4) << this->seconds << " sec" << endl;
```

```
        // Set Decimal Point Precision to four digits below point
    }
    InsertionSort() {
        this->arr = new double[10];
        for (int i = 1; i <= 10; i++) this->arr[i - 1] = i; // Set default array
        this->seconds = clock(); // Set current Millisecond to starting time
    }
    ~InsertionSort() {
        delete this->arr;
    }
    void sorter(double* arr, int size) {
        if (timer()) {
            for (int i = 1; i <= size - 1; i++) {
                double key = arr[i];
                int j = i - 1;
                while (j >= 0 && arr[j] > key) {
                    arr[j + 1] = arr[j];
                    j--;
                } arr[j + 1] = key;
            }
        } else cout << "Terminated due to the time limit" << endl;;
    }
    void run() {
        sorter(this->arr, this->size); // Initiate Recursion
        this->seconds = double(clock() - this->seconds) / double(CLOCKS_PER_SEC); // Elapsed Time in SE
    }
    void set(double *arr, int size) {
        delete[] this->arr;
        this->arr = nullptr;
        this->arr = new double[size];
        for (int i = 0; i < size; i++) this->arr[i] = arr[i];
        this->size = size;
    }
private:
    double *arr;
    int size;
    double seconds;
};

class StoogeSort {
public:
    bool timer() {
```

```
    clock_t current = clock();
    double passed = (double)(current - this->seconds) / CLOCKS_PER_SEC;
    return passed <= 10;
}
void print_time() const {
    cout << "StoogeSort : " << fixed << setprecision(4) << this->seconds << " sec" << endl;
    // Set Decimal Point Precision to four digits below point
}
friend bool check_stooge(StoogeSort*);
StoogeSort() {
    this->arr = new double[10];
    for (int i = 1; i <= 10; i++) this->arr[i - 1] = i; // Set default array
    this->seconds = clock(); // Set current Millisecond to starting time
}
~StoogeSort() {
    delete[] this->arr;
}
void sorter(double* arr, int begin, int end) {
    if (timer()) {
        if (begin >= end) return;
        else if (end - begin == 1) {
            if (arr[begin] > arr[end]) swap(arr[begin], arr[end]);
        }
        else {
            int d = (end - begin + 1) / 3;
            if (!timer()) cout << "Terminated due to the time limit" << endl;
            else sorter(arr, begin, end - d);
            if (!timer()) cout << "Terminated due to the time limit" << endl;
            else sorter(arr, begin + d, end);
            if (!timer()) cout << "Terminated due to the time limit" << endl;
            else sorter(arr, begin, end - d);
        }
    } else cout << "Terminated due to the time limit" << endl;
}
void run() {
    sorter(this->arr, 0, this->size - 1); // Initiate Recursion
    this->seconds = double(clock() - this->seconds) / double(CLOCKS_PER_SEC); // Elapsed Time in SE
}
void set(double *arr, int size) {
    delete[] this->arr;
    this->arr = nullptr;
    this->arr = new double[size];
```

```
        for (int i = 0; i < size; i++) this->arr[i] = arr[i];
        this->size = size;
    }
    void swap(double &a, double &b) {
        double temp = a;
        a = b;
        b = temp;
    }
private:
    double *arr;
    int size;
    double seconds;
};

class HeapSort {
friend bool check_heap(HeapSort *);
public:
    bool timer() {
        clock_t current = clock();
        double passed = (double)(current - this->seconds) / CLOCKS_PER_SEC;
        return passed <= 10;
    }
    void print_time() const {
        cout << "HeapSort : " << fixed << setprecision(4) << this->seconds << " sec" << endl;
        // Set Decimal Point Precision to four digits below point
    }
    HeapSort() {
        this->arr = new double[10];
        for (int i = 1; i <= 10; i++) this->arr[i - 1] = i; // Set default array
        this->seconds = clock(); // Set current Millisecond to starting time
    }
    ~HeapSort() {
        delete this->arr;
    }
    void sorter(double *a, int i, int n) {
        double temp = a[i];
        int j = 2 * i;
        while (j <= n) {
            if (j < n && a[j + 1] > a[j]) j++;
            // Break if parent value is already greater than child value.
            if (temp > a[j]) break;
                // Switching value with the parent node if temp < a[j].
```

```
            else if (temp <= a[j]) {
                a[j / 2] = a[j];
                j *= 2;
            }
        } a[j / 2] = temp;
    }
    void run() {
        for (int i = (this->size - 1) / 2; i >= 1; i--) {
            if (!timer()) {
                cout << "Terminated due to the time limit" << endl;;
                break;
            } sorter(this->arr, i, this->size - 1);
        } if (timer()) {
            for (int i = this->size - 1; i >= 2; i--) {
                if (!timer()) {
                    cout << "Terminated due to the time limit" << endl;;
                    break;
                } swap(this->arr[i], this->arr[1]); // Store maximum value at end
                sorter(this->arr, 1, i - 1); // Build max heap of remaining elements
            }
        }
        this->seconds = double(clock() - this->seconds) / double(CLOCKS_PER_SEC); // Elapsed Time in SE
    }
    void set(double *arr, int size) {
        delete[] this->arr;
        this->arr = nullptr;
        this->arr = new double[size + 1]; arr[0] = 0;
        for (int i = 1; i <= size; i++) this->arr[i] = arr[i];
        this->size = size;
    }
    void swap(double &a, double &b) {
        auto temp = a;
        a = b;
        b = temp;
    }

private:
    double *arr;
    int size;
    double seconds;
};
```

```
#endif
```

# Part II

# C++