

# Student's Manual for Programming Methodology with C++

Paul Kim

powered by L<sup>A</sup>T<sub>E</sub>X 2<sub>ε</sub>

# Contents

**Part I**

**Algorithms**

# Chapter 1

## Time Complexity Analysis

1. The running time of an algorithm is relevant 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:  $T(n) \leq c \times f_O(n) \Rightarrow T(n) = O(f_O(n))$  Best-case scenarios can be described via Big-O functions.
  - 2) Big-Omega:  $T(n) \geq c \times f_\Omega(n) \Rightarrow T(n) = \Omega(f_\Omega(n))$  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,  $T(n) = \max(T_1(n), T_2(n), \dots)$ . Best-case = minimum.
- 3) Recursion:  $T(n) = f(T(n'))$  (점화식)

## Chapter 2

# Finding the Maximum Subarray Sum

The objective of this challenge is to find the maximum value of the sum of elements in a subarray of a given array. If all integers are negative, said maximum value is the sum of a subarray equivalent to the 'empty set', which is zero.

### 2.1 Cubic Brute Force Algorithm

```
6  int max_sum1(int* arr, int arrsize) {
7      int maxSum = 0;
8      for (int i = 0; i < arrsize; i++) {
9          for (int j = i; j < arrsize; j++) {
10             int thisSum = 0;
11             for (int k = i; k <= j; k++) thisSum += arr[k];
12             if (maxSum < thisSum) maxSum = thisSum;
13         }
14     } return maxSum;
15 }
```

This algorithm utilises three loops and a function of constant time in the innermost loop. Therefore, the time complexity analysis goes:

$$T(n) = \sum_{i=1}^n \sum_{j=i}^n \sum_{k=i}^j 1 = O(n^3)$$

Obviously this method is quite wasteful in both memory and timekeeping. The following two algorithms are substantial progressions from this algorithm:

## 2.2 Quadratic Brute Force Algorithm

```

18  int max_sum2(int* arr, int arrsize) {
19      int maxSum = 0;
20      for (int i = 0; i < arrsize; i++) {
21          int iSum = 0;
22          for (int j = i; j < arrsize; j++) {
23              iSum += arr[j];
24              if (maxSum < iSum) maxSum = iSum;
25          }
26      } return maxSum;
27  }

```

This algorithm utilises two loops and a function of constant time in the innermost loop. Therefore, the time complexity analysis goes:

$$T(n) = \sum_{i=1}^n \sum_{j=i}^n 1 = O(n^2)$$

## 2.3 Divide and Conquer

```

30  int max_sum3(int* arr, int left, int right) {
31      if (left >= right) return arr[left];
32      else { int hereMax = 0;
33          int leftSum = max_sum3(arr, left, ((left + right) / 2) - 1);
34          int rightSum = max_sum3(arr, ((left + right) / 2) + 1, right);
35          if (leftSum >= rightSum) hereMax = leftSum;
36          else hereMax = rightSum;
37          int leftMax = arr[(left + right) / 2], leftTemp = 0;
38          int rightMax = arr[(left + right) / 2], rightTemp = 0;
39          for (int i = (left + right) / 2; i >= left; i--) {
40              leftTemp += arr[i];
41              if (leftMax < leftTemp) leftMax = leftTemp;
42          } for (int i = (left + right) / 2; i <= right; i++) {
43              rightTemp += arr[i];
44              if (rightMax < rightTemp) rightMax = rightTemp;
45          } int midSum = leftMax + rightMax - arr[(left + right) / 2];
46          if (hereMax < midSum) hereMax = midSum;
47          return hereMax;
48      }
49  }

```

This algorithm utilises a divisive recursion and selection controls of constant time within each recursion. Therefore, the time complexity analysis goes:

$$T(n) = 2T\left(\frac{n}{2}\right) + n = 2^k T\left(\frac{n}{2^k}\right) + nk, \therefore T(n) = 2^{\log_2 n} T(1) + n \log_2 n = n \log_2 n + \varepsilon n = O(n \log n)$$

While these two algorithms are quite effective compared to Cubic Brute Force, this last algorithm manages to provide the fastest solution possible:

## 2.4 Kadane's Algorithm: A Linear, Incremental Solution

```
52 int max_sum4(int* arr, int arrsize) {  
53     int maxSum = arr[0], thisSum = 0;  
54     for (int i = 0; i < arrsize; i++) {  
55         thisSum += arr[i];  
56         if (thisSum > maxSum) maxSum = thisSum;  
57         else if (thisSum < 0) thisSum = 0;  
58     } return maxSum;  
59 }
```

This algorithm utilises one loop and a function of constant time in the loop. Therefore, the time complexity analysis goes:

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

Since, essentially, the challenge requires a system to read each data at least once, the  $O(n)$  function above is obviously the best solution.

## Chapter 3

# Various Ways of Sorting

The objective of this challenge is sorting given algebraic (in this case, double) elements of an array in increasing order. Some elements may be of equal value.

### 3.1 Quick Sort

```
5     double pivot = arr[begin];
6     int i = begin, j = end;
7     while (i <= j) {
8         while (arr[i] < pivot) i++;
9         while (arr[j] > pivot) j--;
10        if (i <= j) {
11            swap(arr[i], arr[j]);
12            i++; j--;
13        }
14    } if (begin < j) quicksort(arr, begin, j);
15    if (end > i) quicksort(arr, i, end);
16 }
17
87 int main(int argc, char *argv[]) {
88     double *input_array = new double[2500];
89     for (int i = 0; i < 2500; i++) input_array[i] = double(rand() % 2500);
90
91     double *quick_arr = new double[2500];
92     for (int i = 0; i < 2500; i++) quick_arr[i] = input_array[i];
93     quicksort(quick_arr, 0, 2499);
94     if (check(quick_arr)) cout << "Quicksort Validated" << endl;
95
120    return 0;
121 }
```

Quick Sort is a recursive algorithm that does the following:



1. Choose a “pivot” element: Line 6
2. Swap elements that are larger than the pivot with elements that are smaller but on the righthand side of the chosen element: Lines 8:14
3. Continue until all elements are sorted, then recursively proceed with the left and right subarrays of the pivot: Lines 15, 16

Using two half-recursions results in the following time complexity analysis:

$$T(n) = 2T\left(\frac{n}{2}\right) + n = 2^k T\left(\frac{n}{2^k}\right) + kn = O(n \log n)$$

## 3.2 Merge Sort

```

19     if (begin < end) {
20         int centre = begin + (end - begin) / 2;
21         mergesort(arr, begin, centre);
22         mergesort(arr, centre + 1, end);
23         int n1 = centre - begin + 1;
24         int n2 = end - centre;
25         double* L = new double[n1 + 1];
26         double* R = new double[n2 + 1];
27         for (int i = 0; i <= n1 - 1; i++) L[i] = arr[begin + i];
28         for (int j = 0; j <= n2 - 1; j++) R[j] = arr[centre + j + 1];
29         L[n1] = (double)INT_MAX;
30         R[n2] = (double)INT_MAX;
31         int a = 0, b = 0;
32         for (int k = begin; k <= end; k++) {
33             if (L[a] <= R[b]) arr[k] = L[a++];
34             else arr[k] = R[b++];
35         }
36     }
37 }

87 int main(int argc, char *argv[]) {
88     double *input_array = new double[2500];
89     for (int i = 0; i < 2500; i++) input_array[i] = double(rand() % 2500);

96     double *merge_arr = new double[2500];
97     for (int i = 0; i < 2500; i++) merge_arr[i] = input_array[i];
98     mergesort(merge_arr, 0, 2499);
99     if (check(merge_arr)) cout << "Mergesort Validated" << endl;

120     return 0;
121 }
```

Merge Sort is also a recursive algorithm that depends on two half-recursions, so the time complexity analysis goes:

$$T(n) = 2T\left(\frac{n}{2}\right) + n = 2^k T\left(\frac{n}{2^k}\right) + kn = O(n \log n)$$

### 3.3 Insertion Sort

```

46 void insertionsort(double* arr, int size) {
47     for (int i = 1; i <= size - 1; i++) {
48         double key = arr[i];
49         int j = i - 1;
50         while (j >= 0 && arr[j] > key) {
51             arr[j + 1] = arr[j];
52             j--;
53         } arr[j + 1] = key;
54     }
55 }

87 int main(int argc, char *argv[]) {
88     double *input_array = new double[2500];
89     for (int i = 0; i < 2500; i++) input_array[i] = double(rand() % 2500);

101     double *insertion_arr = new double[2500];
102     for (int i = 0; i < 2500; i++) insertion_arr[i] = input_array[i];
103     insertionsort(insertion_arr, 2500);
104     if (check(insertion_arr)) cout << "Insertionsort Validated" << endl;

120     return 0;
121 }

```

Insertion Sort is a double-loop algorithm, therefore time complexity analysis is as follows:

$$T(n) = \sum_{i=1}^n \sum_{j=1}^i \epsilon = O(n^2)$$

### 3.4 Stooge Sort

```

57 void stoogesort(double* arr, int begin, int end) {
58     if (begin >= end) return;
59     else if (end - begin == 1) {
60         if (arr[begin] > arr[end]) swap(arr[begin], arr[end]);
61     } else {
62         int d = (end - begin + 1) / 3;
63         stoogesort(arr, begin, end - d);
64         stoogesort(arr, begin + d, end);
65         stoogesort(arr, begin, end - d);
66     }
67 }

87 int main(int argc, char *argv[]) {
88     double *input_array = new double[2500];
89     for (int i = 0; i < 2500; i++) input_array[i] = double(rand() % 2500);

```

```

106     double *stooge_arr = new double[2500];
107     for (int i = 0; i < 2500; i++) stooge_arr[i] = input_array[i];
108     stoogesort(stooge_arr, 0, 2499);
109     if (check(stooge_arr)) cout << "Stoogesort Validated" << endl;

120     return 0;
121 }

```

Stooge Sort is a recursive algorithm of three subarrays. The time complexity is:

$$T(n) = 3T\left(\frac{3}{2}n\right) + 1, \therefore T(n) \approx O(n^{2.7})$$

### 3.5 Heap Sort

```

69 void heapsort(double *arr, int n, int i) {
70     int largest = i;
71     int l = 2 * i + 1;
72     int r = 2 * i + 2;
73     if (l < n && arr[l] > arr[largest]) largest = l;
74     if (r < n && arr[r] > arr[largest]) largest = r;
75     if (largest != i) {
76         swap(arr[i], arr[largest]);
77         heapsort(arr, n, largest);
78     }
79 }

87 int main(int argc, char *argv[]) {
88     double *input_array = new double[2500];
89     for (int i = 0; i < 2500; i++) input_array[i] = double(rand() % 2500);

111     double *heap_arr = new double[2500];
112     for (int i = 0; i < 2500; i++) heap_arr[i] = input_array[i];
113     for (int i = 1249; i >= 0; i--)
114         heapsort(heap_arr, 2500, i);
115     for (int i = 2499; i >= 0; i--) {
116         swap(heap_arr[0], heap_arr[i]);
117         heapsort(heap_arr, i, 0);
118     } if (check(heap_arr)) cout << "Heapsort Validated" << endl;

120     return 0;
121 }

```

Heap sort is a loop-and-recursive algorithm of time complexy analysis

$$T(n) = O(n \log n)$$

## **Part II**

## **C++**

# Chapter 1

## Variables and Class

### 1.1 The Basics

```
1  #include <iostream>
2
3  int main(void) {
4      std::cout << "Hello World!";
5      return 0;
6  }
```

### 1.2 $j++i$

### 1.3 $j++i$

### 1.4 $j++i$

## Chapter 2

# Functions

2.1  $j++i$

2.2  $j++i$

2.3  $j++i$

2.4  $j++i$

## Chapter 3

# iostream Headers

3.1 `j++i`

3.2 `j++i`

3.3 `j++i`

3.4 `j++i`

## Chapter 4

# Arrays and Pointers

4.1 `j++i`

4.2 `j++i`

4.3 `j++i`

4.4 `j++i`



## Chapter 5

# Object-Orientated Programming

5.1 `j++i`

5.2 `j++i`

5.3 `j++i`

5.4 `j++i`

## Chapter 6

# Defining Classes with OOP

6.1 `j++i`

6.2 `j++i`

6.3 `j++i`

6.4 `j++i`

## Chapter 7

# Member Functions

7.1 `j++i`

7.2 `j++i`

7.3 `j++i`

7.4 `j++i`

## Chapter 8

# Namespace and STL

8.1 `j++i`

8.2 `j++i`

8.3 `j++i`

8.4 `j++i`

## Chapter 9

# Constructors and Destructors

9.1 `j++i`

9.2 `j++i`

9.3 `j++i`

9.4 `j++i`

## Chapter 10

# Public or Private, Friend Declarations

10.1 `i++i`

10.2 `i++i`

10.3 `i++i`

10.4 `i++i`

## Chapter 11

# Copy Constructors

11.1 `i++i`

11.2 `i++i`

11.3 `i++i`

11.4 `i++i`

## Chapter 12

# Operator Overloading and the Rule of Three

12.1 `i++i`

12.2 `i++i`

12.3 `i++i`

12.4 `i++i`



## Chapter 13

# Protected and Private Derivations

13.1  $j++i$

13.2  $j++i$

13.3  $j++i$

13.4  $j++i$

## Chapter 14

# Virtual Functions

14.1 `j++i`

14.2 `j++i`

14.3 `j++i`

14.4 `j++i`

## Chapter 15

# Pure Virtual Functions

15.1 `j++i`

15.2 `j++i`

15.3 `j++i`

15.4 `j++i`

## Chapter 16

# Reusing Copy Control Members

16.1 `i++i`

16.2 `i++i`

16.3 `i++i`

16.4 `i++i`