Student's Manual for Programming Methodology

with C++

Paul Kim

powered by $\LaTeX\,2_\varepsilon$

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

Objective 1: 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

```
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;
}
```

2.2 Quadratic Brute Force Algorithm

```
//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;
}
```

2.3 Divide and Conquer

```
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;
    }
}
```

2.4 Kadane's Algorithm: A Linear, Incremental Solution

```
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;
}
```

Part II

C++