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: $T(n) \geq c \times f_O(n) \;\; \Rightarrow \;\; T(n) = O\left(f_O(n)\right)$ Best-case scenarios can be described via Big-O functions.

   2) Big-Omega: $T(n) \leq c \times f_\Omega(n) \;\; \Rightarrow \;\; T(n) = \Omega\left(f_\Omega(n)\right)$ 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, $T(n) = \max\left(T_1(n),\ T_2(n), \cdots\right)$. 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

```
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;
}
```

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

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

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

2.2 Quadratic Brute Force 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;
}
```

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

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

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;
    }
}
```

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

```
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;
}
```

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 callenge is sorting given algebraic (in this case, `double`) elements of an aray in incrasing order. Some elements may be of equal value.

3.1 Quick Sort

```
    double pivot = arr[begin];
    int i = begin, j = end;
    while (i <= j) {
        while (arr[i] < pivot) i++;
        while (arr[j] > pivot) j--;
        if (i <= j) {
            swap(arr[i], arr[j]);
            i++; j--;
        }
    } if (begin < j) quicksort(arr, begin, j);
    if (end > i) quicksort(arr, i, end);
}

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

    double *quick_arr = new double[2500];
    for (int i = 0; i < 2500; i++) quick_arr[i] = input_array[i];
    quicksort(quick_arr, 0, 2499);
    if (check(quick_arr)) cout << "Quicksort Validated" << endl;

    return 0;
}
```

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

```
    if (begin < end) {
        int centre = begin + (end - begin) / 2;
        mergesort(arr, begin, centre);
        mergesort(arr, centre + 1, 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, b = 0;
        for (int k = begin; k <= end; k++) {
            if (L[a] <= R[b]) arr[k] = L[a++];
            else arr[k] = R[b++];
        }
    }
}
```

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

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

    return 0;
}
```

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

```
void insertionsort(double* arr, int size) {
    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;
    }
}

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

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

    return 0;
}
```

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

```
void stoogesort(double* arr, int begin, int end) {
    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;
        stoogesort(arr, begin, end - d);
        stoogesort(arr, begin + d, end);
        stoogesort(arr, begin, end - d);
    }
}

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

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

    return 0;
}
```

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

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

3.5 Heap Sort

```
void heapsort(double *arr, int n, int i) {
    int largest = i;
    int l = 2 * i + 1;
    int r = 2 * i + 2;
    if (l < n && arr[l] > arr[largest]) largest = l;
    if (r < n && arr[r] > arr[largest]) largest = r;
    if (largest != i) {
        swap(arr[i], arr[largest]);
        heapsort(arr, n, largest);
    }
}

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

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

    return 0;
}
```

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

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

Part II

C++