CH08-320143: Assignment #4

Harit Krishan
h.krishan@jacobs-university.de

Jacobs University Bremen — March 2, 2020

1 Merge Sort

Merge Sort is a divide and conquer algorithm. It divides the input array into two halves, and then recursively calls itself on those two halves until we have arrays of size 1. At this point, we merge the two sorted halves. This algorithm works in two parts - array division & array merging. We define two different functions. One function breaks the problem down into sub-problems of size 1, and the other merges the sub-solutions to output the global solution. Code can be modified such that the problem is broken down to sub-problems of a particular size k. The following sub-section shows the implementation of this variant of Merge Sort in C++.

1.1 Implementation

As mentioned earlier, Merge Sort breaks the problem down into sub-problems. Figure 1 shows the code for dividing the problem down into sub-problems of size k.

```
void merge_sort(int *arr, int low, int high, int k) {

        if(((high - low) + 1) <= k) {
                insertion_sort(arr, low, high);
        } else {
                int mid;
                if(low < high) {
                        mid = (low + high)/2;
                        merge_sort(arr, low, mid, k);
                        merge_sort(arr, mid+1, high, k);
                        merge(arr, low, high, mid);
                }
        }
}
```

Figure 1: Implementation of Merge Sort in C++

We see that the function $merge_sort()$ takes as arguments the array pointer, the low and high indices, and value k. We use this k as a threshold parameter of the lengths of the sub-arrays. If the length of the sub-arrays are less than or equal to this threshold parameter, then Insertion Sort will be applied on those sub-arrays. Otherwise, the algorithm further breaks down the array into halves by calling itself on the sub-arrays. Once global problem has been broken down to the desired-sized sub-arrays, they are merged using the $merge()$ function. Figure 2 shows the implementation of the $merge()$ function in C++.

In the $merge()$ function from Figure 2, the algorithm initializes a second array which will be used to store all the elements in the right order. We create a variable to iterate the left sub-array, a variable to iterate the right sub-array, and a variable to iterate the output array. The output array iterator will iterate regardless of any conditions. However, the other two variables will iterate with respect to the respective values in the respective arrays. The lesser of the two values will be put into the output array to achieve the increasing order. If either of the sub-arrays have not been completely extracted, the ramaining blocks of

code achieve that in the implied order - first completely extract the left sub-array, then the right sub-array. Once the key extraction is complete from all sub-arrays, the original array is corrected by syncing with the corrected array. Hence, the loop invariant is conserved.

```
void merge(int *arr, int low, int high, int mid) {
        int i, j, k;
        int arrb[10000];
        i = k = low;
        j = mid + 1;

        while(i <= mid && j <= high) {                  // Iterate both sub-arrays
                if(arr[i] < arr[j]) {                   // Lesser of the two selected
                        arrb[k] = arr[i];
                        k++;
                        i++;
                } else {
                        arrb[k] = arr[j];
                        k++;
                        j++;
                }
        }

        while(i <= mid) {                       // Remaining (if any) from left sub-array
                arrb[k] = arr[i];
                k++;
                i++;
        }

        while(j <= high) {                      // Remaining (if any) from right sub-array
                arrb[k] = arr[j];
                k++;
                j++;
        }

        for(i = low; i < k; i++) {              // Sorted sequnce put inplace of input array
                arr[i] = arrb[i];
        }
}
```

Figure 2: Implementation of Merge Function in C++

1.2 Runtime Case Analysis

In runtime analysis, we look at the best, worst and average cases, ignoring machine-dependent constants. The worst case for this algorithm would be if the input array is in decreasing order. A worst case sequence can be generated using the following line of code -

$$arr[i] = size - i$$

The best case input for this algorithm would be if the input array is already sorted. Therefore, to time this case, we can first sort a random sequence using any algorithm, and then pass it on to Merge Sort. We time the execution of Merge Sort to get the best case runtime estimate. This would be achieved by adding the following line of code before timing the algorithm -

$$merge\_sort(arr,\ 0,\ size - 1,\ k)$$

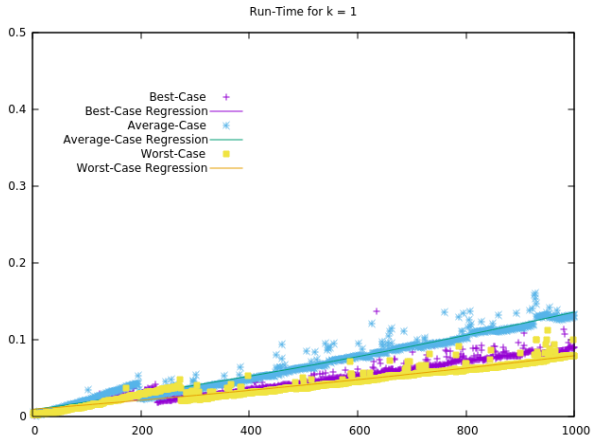
The average case input for any algorithm can be considered to be any random input. To achieve this we make use of the $rand()$ function in C++. First we must initialize the seed using the $srand()$ function. To generate an average case array, we use the following lines of code to generate random numbers in the range 1 to 99 -

$$srand(time(NULL));$$

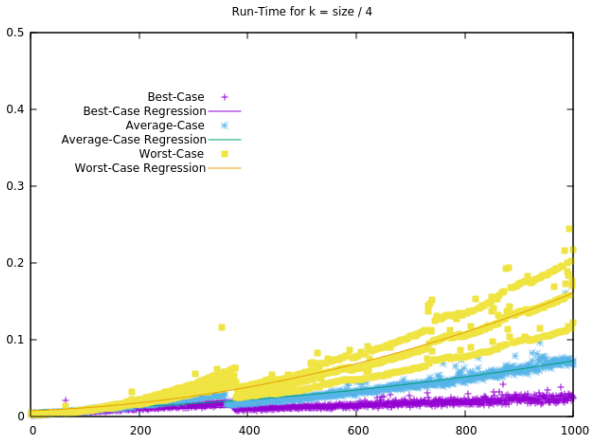
$$random\_number\ =\ rand()\ \%\ 99\ +\ 1;$$

The above-described cases have been implemented into the driver function to generate the respective input arrays. The algorithm has then been tested and timed on these cases for varying sizes of input arrays with 4 different values for k being 1, $size/2$, $size/4$ and $size - 1$.

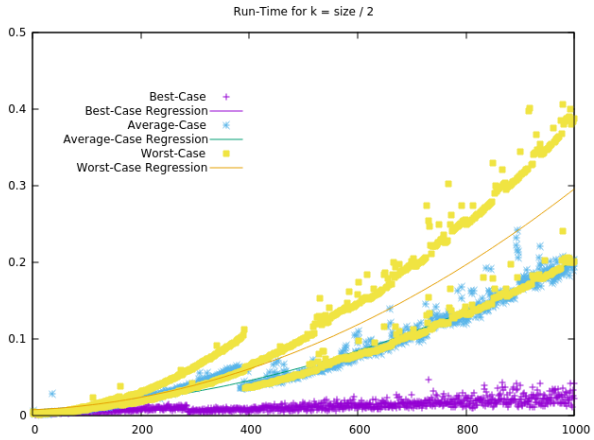
The table below shows the run-time vs. input size graphs for the varying values of k.



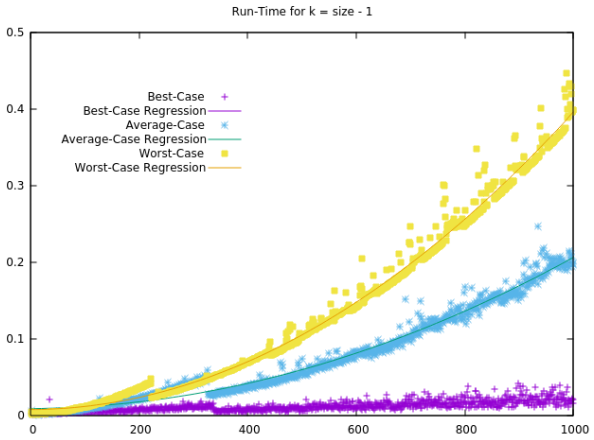
(a) Insertion Sort applied to sub-arrays of size 1



(b) Insertion Sort applied to sub-arrays of size n/4



(c) Insertion Sort applied to sub-arrays of size n/2



(d) Insertion Sort applied to sub-arrays of size n - 1

Figure 3: Graphs of Running-Times for Varying Values of K for Each Case

1.3 Varying k

As the value of k increases, it means that larger sub-arrays will be sorted using Insertion Sort. As evident from Figure 3, one can tell that for the average case, at each given input size, the running time of the algorithm is greater. This happens because Insertion Sort has a greater time complexity than Merge Sort for randomly arranged arrays.

However, for the best case, we pass a sorted array as a parameter into our custom sorting function. Increasing k for the best case reduces the computation time of the algorithm. This happens because insertion sort compares the current element with the previous to make sure the order is increasing. This means the algorithm (Insertion Sort) will have traversed the entire array and the only executed operation will be

the comparison of the two initial values.

Lastly, for the worst case, we observe a similar trend as the average case. As k increases, we get a higher algorithm run time.

1.4 Algorithm in Practice

Insertion Sort is most efficient on smaller arrays. This implies that as the size of the array increases, the run time of Insertion Sort will get longer, rendering the algorithm slow. Merge Sort is faster than Insertion Sort in that regard, therefore the most efficient implementation of this custom algorithm would be to choose a particular combination of the two algorithms. Ideally, we would want to keep the value of k low to get faster computation times.

2 Recurrences

We use the substitution method, recursion tree method & master theorem to solve recurrences.

2.1 $T(n) = 36T(n/6) + 2n$

It would be simple to apply the Master's method to solve this recurrence. We know that recurrences are written in the following form:

$$\begin{aligned} T(n) &= aT(n/b) + f(n) \\ T(n) &= 36T(n/6) + 2n \end{aligned}$$

From the above equations, we generalize the following constants:

$$a = 36, \quad b = 6, \quad f(n) = 2n,$$

We use the following identity to solve for the exponent x':

$$\begin{aligned} n^{\log_b a} &= n^{x'} \\ \log_b a &= x' \\ 2 &= x' \end{aligned}$$

If we take ϵ to be 0.2:

$$\begin{aligned} n^{2-\epsilon} &= n^{1.8} \\ \lim_{n\to\infty} \frac{2n}{n^{1.8}} &= \lim_{n\to\infty} \frac{2}{n^{0.8}} \\ &= 0 \end{aligned}$$

The above implies the following:

$$\begin{aligned} f(n) = O(g(n)) = O(n^{2-\epsilon}) = 2n \\ \therefore \quad T(n) = \Theta(n^2) \end{aligned}$$

2.2 $T(n) = 5T(n/3) + 17n^{1.2}$

We make use of the same technique to solve this recurrence as the last problem. We know that recurrences are written in the following form:

$$\begin{aligned} T(n) &= aT(n/b) + f(n) \\ T(n) &= 5T(n/3) + 17n^{1.2} \end{aligned}$$

From the above equations, we generalize the following constants:

$$a = 5, \quad b = 3, \quad f(n) = 17n^{1.2},$$

We use the following identity to solve for the exponent x':

$$\begin{aligned} n^{\log_b a} &= n^{x'} \\ \log_b a &= \log_3 5 = x' \\ 1.46 &= x' \end{aligned}$$

If we take ϵ to be 0.2:

$$\begin{aligned} n^{1.46-\epsilon} &= n^{1.26} \\ \lim_{n\to\infty} \frac{17n^{1.2}}{n^{1.26}} &= \lim_{n\to\infty} \frac{17}{n^{0.06}} \\ &= 0 \end{aligned}$$

The above implies the following:

$$\begin{aligned} f(n) &= O(g(n)) = O(n^{1.46-\epsilon}) \\ f(n) &= 17n^{1.2} = O(n^{1.46-\epsilon}) \\ f(n) &= O(n^{\log_b a-\epsilon}) \\ \therefore \quad T(n) &= \Theta(n^{\log_b a}) = \Theta(n^{\log_3 5}) \end{aligned}$$

2.3 $T(n) = 12T(n/2) + n^2 \lg n$

We know that recurrences are written in the following form:

$$\begin{aligned} T(n) &= aT(n/b) + f(n) \\ T(n) &= 12T(n/2) + n^2 \lg n \end{aligned}$$

From the above equations, we generalize the following constants:

$$a = 12, \quad b = 2, \quad f(n) = n^2 \lg n,$$

We use the following identity to solve for the exponent x':

$$\begin{aligned} n^{\log_b a} &= n^{x'} \\ \log_b a &= \log_2 12 = x' \\ 3.58 &= x' \end{aligned}$$

If we take ϵ to be 0.2:

$$\begin{aligned} n^{3.58-\epsilon} &= n^{3.38} \\ \lim_{n\to\infty} \frac{n^2 \lg n}{n^{3.38}} &= \lim_{n\to\infty} \frac{\lg n}{n^{1.38}} \\ &= 0 \end{aligned}$$

The above implies the following:

$$\begin{aligned} f(n) &= O(g(n)) = O(n^{3.58-\epsilon}) \\ f(n) &= n^2 \lg n \\ f(n) &= O(n^{\log_b a-\epsilon}) \\ \therefore \quad T(n) &= \Theta(n^{\log_b a}) = \Theta(n^{\lg 12}) \end{aligned}$$

2.4 $T(n) = 3T(n/5) + T(n/2) + 2^n$

To solve this recurrence, we make use of the substitution method. We begin by assuming that $T(k) \leq ck$, where c is a constant.

$$
\begin{array}{rcl}
T(n/5) & < & T(n/2) \\
3T(n/5) & + & T(n/2) < 4T(n/2) \\
4T(n/2) & + & 2^n \leq 4c\frac{n}{2} + 2^n \\
4T(n/2) & + & 2^n \leq 2cn + 2^n
\end{array}
$$

After the above computations, we can say that $T(n) \leq 2cn + 2^n$. The time complexity of $2cn + 2^n$ is$\Theta 2^n$which implies the following:

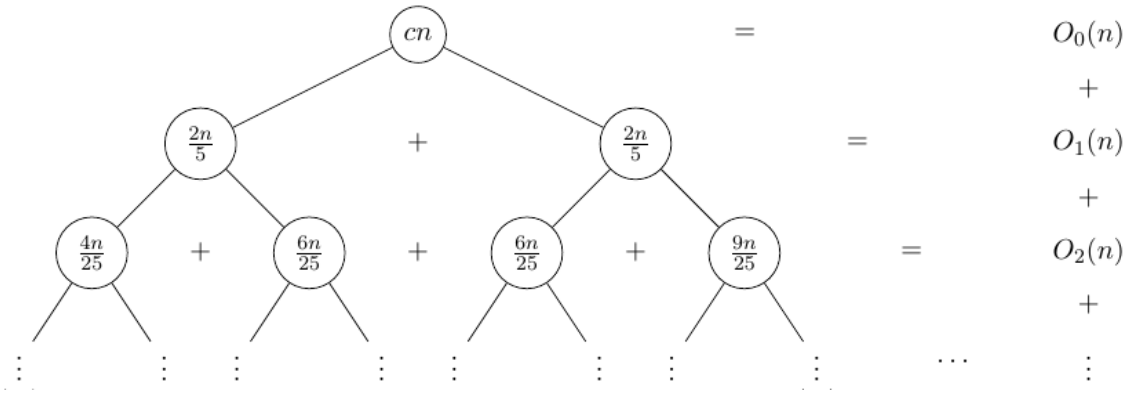
$$\therefore \quad T(n) = \Theta(2^n)$$

2.5 $T(n) = T(2n/5) + T(3n/5) + \Theta(n)$

To solve this recurrence, we make use of the recursion tree method. We begin by defining the following:

$$f(n) = \Theta(n) \Rightarrow f(n) = cn$$

Now we draw the recursion tree. The recursion tree would look something like this:



In the above diagram, we see on the right hand side the summation of time complexities line-wise. If we continue doing so, we get a general formula for the time complexity. The resulting time complexity is:

$$
\begin{array}{rcl}
Total & = & cn \log_{5/3} n \\
\therefore \quad T(n) & = & O(n \lg n)
\end{array}
$$

3 Root Directory Walkthrough

The root directory of this submission zip contains the data files generated for each of the cases for the four different values of k. These files end in .dat and the first two characters signify which case it is - best case (bc*.dat), average case (ac*.dat), worst case(wc*.dat). Graphs have been generated using Gnuplot. The script for this program is also available in this directory by the name $gnuplot_script$. In this script, we first compute the respective regressions and graph them alongside the raw data from the data files. The pdf file consists of all the solutions with snapshots of the code. The algorithm implementation is in the file $merge_sort.cpp$.