# Algorithms and Data Structures: Homework #4

Due on March 2, 2020 at 23:00 PM

**Henri Sota**

# Problem 4.1

a) Implement a variant of Merge Sort that does not divide the problem all the way down to subproblems of size 1. Instead, when reaching subsequences of length $k$ it applies Insertion Sort on these $n/k$. subsequences.

**Algorithm 1** Merge Sort[A, p, r, k]

**if** $r - p + 1 <= k$ **then**
  Insertion Sort$(A, p, r)$
**else**
  $q = (p + r)/2$
  Merge Sort$(A, p, q, k)$
  Merge Sort$(A, q + 1, r, k)$
  Merge$(A, p, q, r)$
**end if**

**Algorithm 2** Insertion Sort[A, p, r]

**for** $j = p + 1$ **to** $r$ **do**
  $key = A[j]$
  $i = j - 1$
  **while** $i > p - 1$ and $A[i] > key$ **do**
    $A[i + 1] = A[i]$
    $i = i - 1$
  **end while**
  $A[i + 1] = key$
**end for**

**Algorithm 3** Merge[A, p, q, r]

$n_1 = q - p + 1$
$n_2 = r - q$
**Declare** $L$: Array with $n_1 + 1$ elements
**Declare** $R$: Array with $n_2 + 1$ elements
**for** $i = 1$ **to** $n_1$ **do**
  $L[i] = A[p + i - 1]$
**end for**
**for** $j = 1$ **to** $n_2$ **do**
  $R[j] = A[q + j]$
**end for**
$L[n_1 + 1] = \infty$
$R[n_2 + 2] = \infty$
$i = 1$
$j = 1$
**for** $k = p$ **to** $r$ **do**
  **if** $L[i] \leq R[j]$ **then**
    $A[k] = L[i]$
    $i = i + 1$
  **else**
    $A[k] = R[j]$
    $j = j + 1$
  **end if**
**end for**

```
void merge(std::vector<int> &arr, int p, int q, int r) {
    std::vector<int> left, right;
    // Fill left vector with elements from p to q of arr
    for (int i = p; i <= q; i++)
        left.push_back(arr[i]);
    // FIll right vector with elements from q+1 to r of arr
    for (int i = q + 1; i <= r; i++)
        right.push_back(arr[i]);
    // Add sentinel value interval as biggest element of each array
    left.push_back(interval);
    right.push_back(interval);
    // Merge both subarrays into one by comparing left[i] and right[j] and
    // choosing the smaller one first
    int i = 0, j = 0;
    for (int k = p; k <= r; k++) {
        if (left[i] <= right[j]) {
            arr[k] = left[i];
            i++;
        } else {
            arr[k] = right[j];
            j++;
        }
    }
}

void insertionSort(std::vector<int> &arr, int p, int r) {
    for (int j = p + 1; j <= r; j++) {
        int key = arr[j];
        int i = j - 1;
```

```
        // Search for the position where to place key value in the sorted
        // subarray arr[0..i]
        while (i > p - 1 && arr[i] > key) {
            arr[i + 1] = arr[i];
            i--;
        }
        arr[i + 1] = key;
    }
}

void mergeSort(std::vector<int> &arr, int p, int r, int k) {
    if (r - p + 1 <= k) {
        // Sort using Insertion Sort
        insertionSort(arr, p, r);
    } else {
        // Sort using Merge Sort
        // Calculate middle position of the array
        int q = (p + r - 1) / 2;
        // Recursively call Merge Sort on the subarray left to middle position
        mergeSort(arr, p, q, k);
        // Recursively call Merge Sort on the subarray right to middle position
        mergeSort(arr, q + 1, r, k);
        // Merge both sorted subarrays into one sorted subarray
        merge(arr, p, q, r);
    }
}
```

Listing 1: Implementation of Merge Sort Optimized with Insertion Sort in C++

b) Apply it to the different sequences which satisfy best case, worst case and average case for different values of $k$. Plot the execution times for different values of $k$.
The results for the following graphs have been calculated for three different cases, when number of elements is 25000, 50000 and 10000, using mergeSortInsertionSortOptimized.cpp with compiling command: g++ -Wall -O2 -o mergeSortInsertionSortOptimized mergeSortInsertionSortOptimized.cpp. Runtime of the optimized algorithm has been calculated for different values of $k$, ranging from 10 to 1000, for each of the three cases mentioned above. The data has been graphed using gnuplot with three gnuplot files found inside **data** folder. Gnuplot code has been modified from the existing one found here.

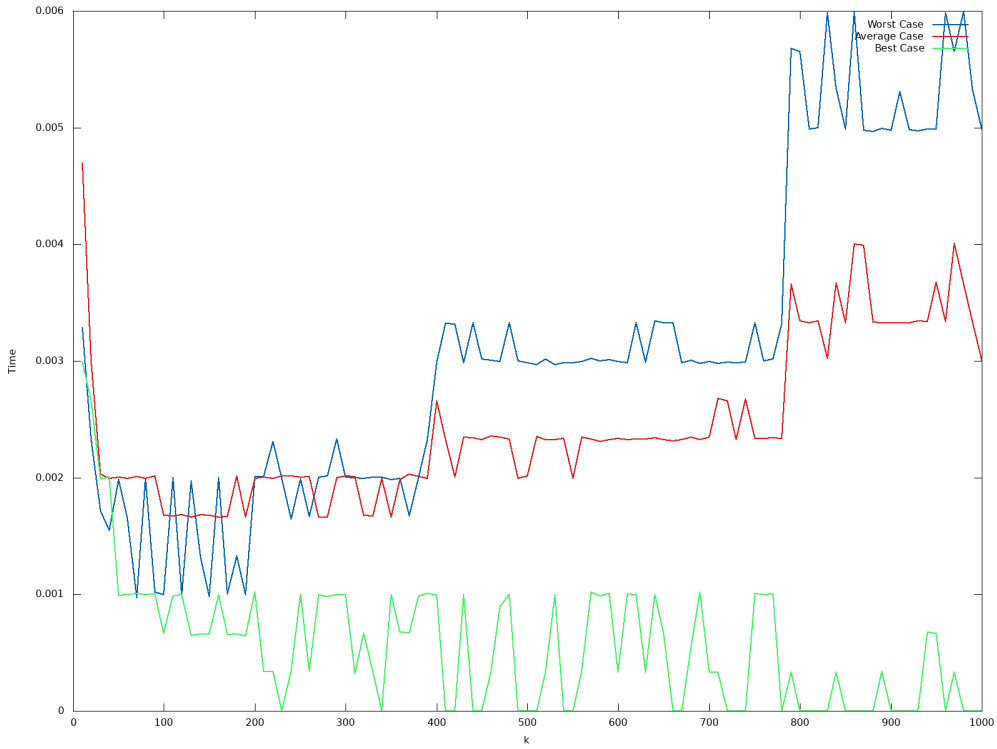



Figure 1: Merge Sort optimized with Insertion Sort plot with different $k$ values and array size 25000

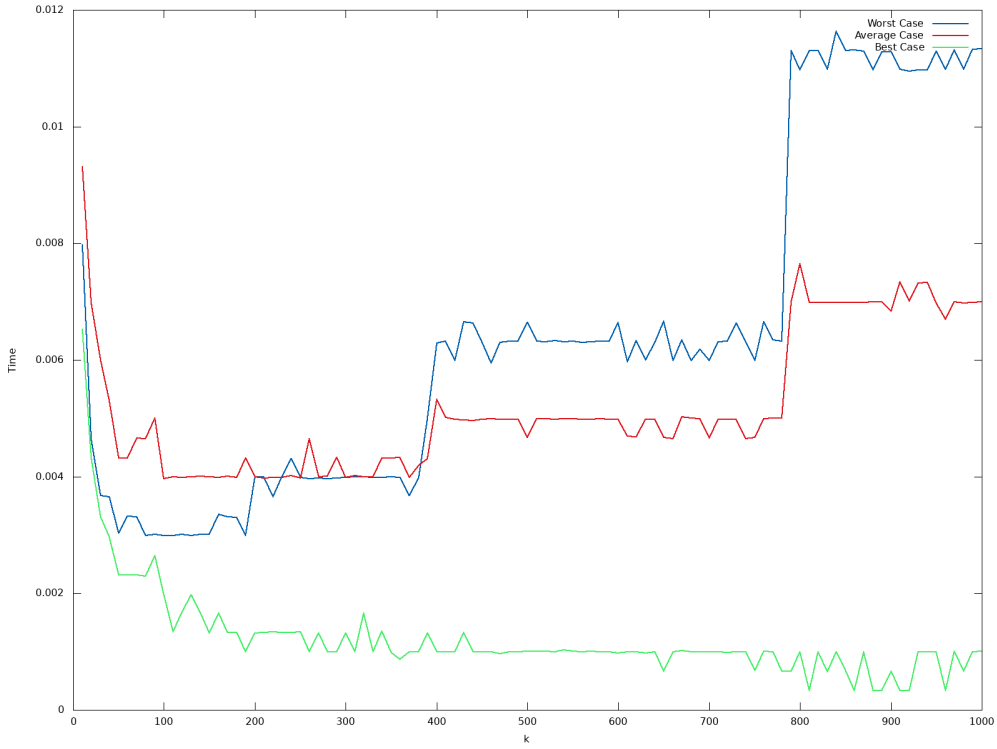



Figure 2: Merge Sort optimized with Insertion Sort plot with different $k$ values and array size 50000

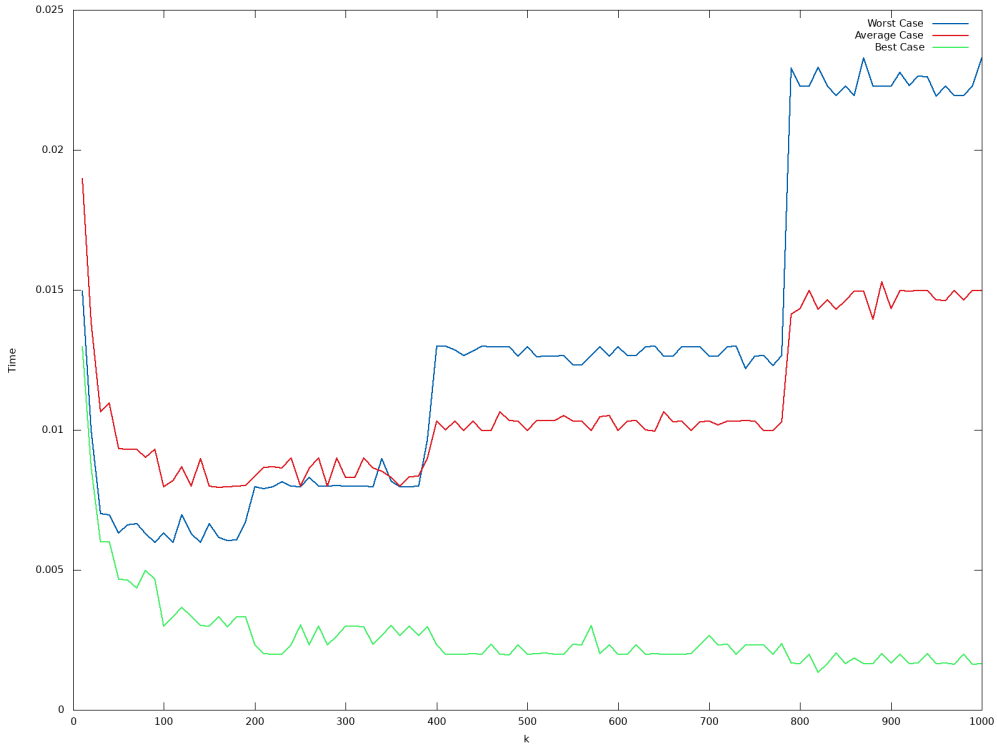



Figure 3: Merge Sort optimized with Insertion Sort plot with different $k$ values and array size 100000

c) How do the different values of $k$ change the best-, average-, and worst-case asymptotic time complexities for this variant? Explain/prove your answer.

- **Worst Case**
  The worst case time to sort a list of length $k$ by insertion sort is $\Theta(k^2)$. Sorting $\frac{n}{k}$ sublists, where each of them is of length $k$ takes $\Theta(k^2 \cdot \frac{n}{k}) = \Theta(nk)$ time. Merging these $\frac{n}{k}$ sorted sublists into a single sorted list of length $n$ will result in $\log(\frac{n}{k})$ steps. Therefore, worst case time to merge the sublists is $\Theta(n \log(\frac{n}{k}))$.
  Merge Sort optimized with Insertion Sort has time complexity as standard merge sort when $\Theta(nk + n\log(\frac{n}{k})) = \Theta(n \log n)$. Assuming $k = \Theta(\log n)$:

$$\begin{aligned}\Theta(nk + n\log(\frac{n}{k})) &= \Theta(nk + n\log n - n\log k) \\ &= \Theta(n\log n + n\log n - n\log(\log n)) \\ &= \Theta(2n\log n - n\log(\log n)) \\ &= \Theta(n\log n)\end{aligned}$$

- **Average Case** Average case runtime is smaller by a constant factor than the worst case runtime, as some of the elements might be in their proper position during sorting with Insertion Sort. The growth rate is the and there is a resemblance between both lines, because as $k$ grows, their complexity time for unsorted arrays of elements is $\Theta(n^2)$.
- **Best Case**
  From the growth rate of the line of the best case in all 3 graphs, the algorithm tends to get faster with higher values of $k$. This is due to the best case complexity of Insertion Sort, which is $O(n)$ (linear time), is smaller than the best case complexity of Merge Sort, which is $O(n \log n)$.

# Problem 4.2

Use the substitution method, the recursion tree, or the master theorem method to derive upper and lower bounds for $T(n)$ in each of the following recurrences. Make the bounds as tight as possible. Assume that $T(n)$ is constant for $n \leq 2$.

a) $T(n) = 36T(n/6) + 2n$
s Master method can be used:
$a = 36$
$b = 6$

$$\begin{aligned} n^{\log_6 36} &= n^2 \\ f(n) &= 2n \\ f(n) &= O(n^{\log_6 36 - \epsilon}) = O(n^{2-\epsilon}) \text{ where } \epsilon = \log_6 36 - 1 = 1 \end{aligned} \tag{1}$$

**Case 1:** $f(n)$ is polynomially smaller than $n^{\log_6 36}$
Result: $T(n) = \Theta(n^{\log_6 36}) = \Theta(n^2)$

b) $T(n) = 5T(n/3) + 17n^{1.2}$
Master method can be used:
$a = 5$
$b = 3$

$$\begin{aligned} n^{\log_3 5} &= n^{1.465} \\ f(n) &= 17n^{1.2} \\ f(n) &= O(n^{\log_3 5 - \epsilon}) = O(n^{1.465-\epsilon}) \text{ where } \epsilon = \log_3 5 - 1.2 = 0.265 \end{aligned} \tag{2}$$

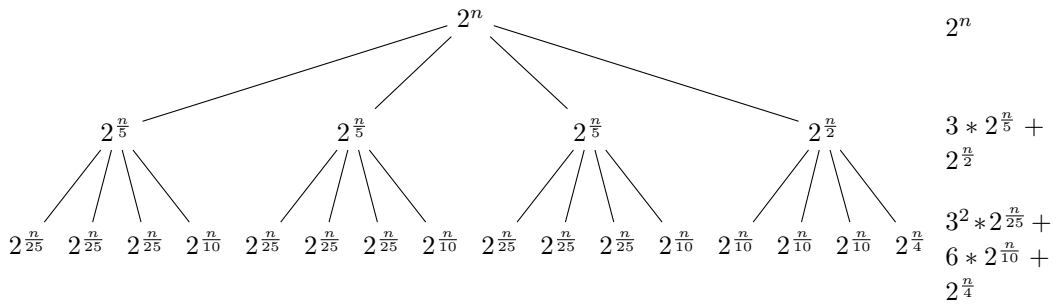
**Case 1:** $f(n)$ is polynomially smaller than $n^{\log_3 5}$
Result: $T(n) = \Theta(n^{\log_3 5}) \approx \Theta(n^{1.465})$

c) $T(n) = 12T(n/2) + n^2 \log n$
Master method can be used:
$a = 12$
$b = 2$

$$\begin{aligned} n^{\log_2 12} &= n^{3.585} \\ f(n) &= n^2 \log n \\ f(n) &= O(n^{\log_2 12 - \epsilon}) = O(n^{3.585-\epsilon}) \text{ where } \epsilon \in \mathbb{R} \end{aligned} \tag{3}$$

**Case 1:** $f(n)$ is asymptotically smaller and polynomially smaller than $n^{\log_2 12}$, because $n^{3.585} > n^2$ and $\Theta(n) \gg \Theta(\log n)$
Result: $T(n) = \Theta(n^{\log_3 5}) \approx \Theta(n^{3.585})$

d) $T(n) = 3T(n/5) + T(n/2) + 2^n$
Recursion tree method can be used:




In order to calculate a tight bound, the height of the tree has to be calculated based on the longest path and the shortest path from the root to a leaf. In this case, the longest path is calculated by following the rightmost node on each level and the shortest path is calculated by the following the leftmost node on each level.

$$h_l = \log_2 n = \frac{\log n}{\log 2} = c_l \log n = O(\log n)$$
$$h_s = \log_5 n = \frac{\log n}{\log 5} = c_s \log n = \Omega(\log n)$$

Each level of the recursion tree has a decreasing cost from the level above it.

$$\begin{aligned}
\text{First Level Cost} &= 2^n \\
\text{Second Level Cost} &= 2^n(3 * 2^{\frac{-4n}{5}} + 2^{\frac{-n}{2}}) \\
\text{Third Level Cost} &= 2^n(3^2 * 2^{\frac{-24n}{25}} + 6 * 2^{\frac{-9n}{10}} + 2^{\frac{-3n}{4}}) \\
&\vdots
\end{aligned}$$

As each level has a decreasing cost, the level with the most weight is the root level, with cost $2^n$. This level will determine the asymptotic behaviour of the overall algorithm.
Induction hypothesis for the substitution method is: $\Theta(2^n)$

$$\begin{aligned}
T(n) &\leq 3c2^{\frac{n}{5}} + c2^{\frac{n}{2}} + 2^n \\
&\leq 3c2^n + c2^n + 2^{n\,[1]} \\
&\leq c2^n
\end{aligned}$$

[1] Known by the fact that $2^n \gg 2^{n/d}$ for $d > 1$
$T(n) = \Theta(2^n)$