Algorithms and Data Structures: Homework 4

Due on March 6th, 2023, 23:00

Problem 4.1: Merge Sort

(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)

```
n1 = q − p + 1, n2 = r − q
Declare L: Array with n1 + 1 elements
Declare R: Array with n2 + 1 elements
for i = 1 to n1 do
  L[i] = A[p + i − 1]
end for
for j = 1 to n2 do
  R[j] = A[q + j]
end for
L[n1 + 1] = ∞, R[n2 + 2] = ∞, i = 1, j = 1
for k = p to r do
  if L[i] ≤ R[j] then
    A[k] = L[i]
    i = i + 1
  else
    A[k] = R[j]
    j = j + 1
  end if
end fo
```

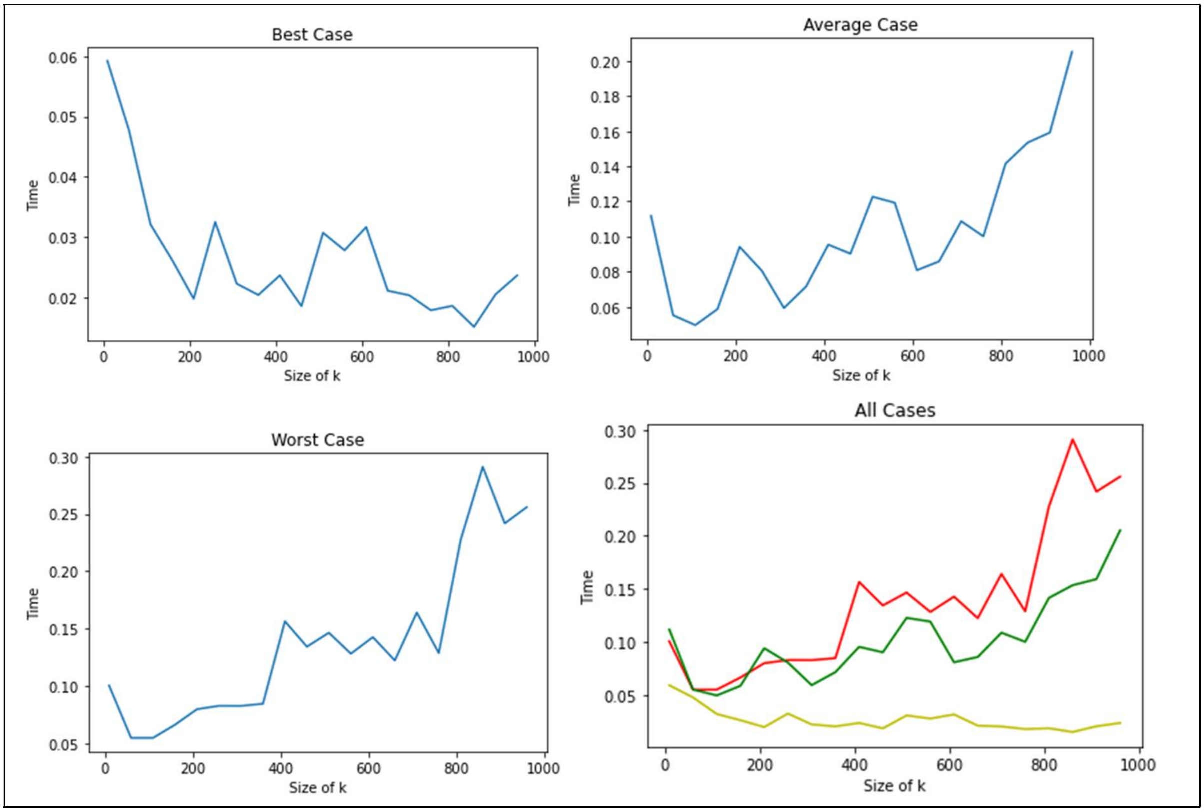
```cpp
void merge(vector<int> &arr, int p, int q, int r) {
    vector<int> left, right;
    for (int i = p; i <= q; i++)
        left.push_back(arr[i]);
    for (int i = q + 1; i <= r; i++)
        right.push_back(arr[i]);
    left.push_back(interval);
    right.push_back(interval);
    int i = 0, j = 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(vector<int> &arr, int p, int r) {
    for (int j = p + 1; j <= r; j++) {
        int key = arr[j];
        int i = j - 1;
        while (i > p - 1 && arr[i] > key) {
            arr[i + 1] = arr[i];
            i--;
        }
        arr[i + 1] = key;
    }
}

void mergeSort(vector<int> &arr, int p, int r, int k) {
    if (r - p + 1 <= k) {
        insertionSort(arr, p, r);
    } else {
        int q = (p + r - 1) / 2;
        mergeSort(arr, p, q, k);
        mergeSort(arr, q + 1, r, k);
        merge(arr, p, q, r);
    }
}
```

Implementation of the algorithms 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.



(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 for Insertion Sort is $\Theta\ (k^2)$. Sorting $\frac{n}{k}$ subarrays, 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 subarrays into a single sorted array of length n will result in $\log(\frac{n}{k})$ steps. Therefore, worst case time to merge the subarrays is $\Theta(n \log(\frac{n}{k}))$.

- 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.

- 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 $O(n \log n)$.

(d) Based on the results from (b) and (c), how would you choose k in practice? Briefly explain.

In order for Merge Sort optimized with Insertion Sort algorithm to perform better than standard Merge Sort, k must be chosen in a way that runtime of the hybrid algorithm to be always lower than the runtime of standard Merge Sort. Insertion Sort running on n k subarrays should be faster than Merge Sort running on those subarrays. Assuming k = Θ(log n): Θ(nk + n log(n k)) = Θ(nk + n log n − n log k) = Θ(n log n + n log n − n log(log n)) = Θ(2n log n − n log(log n)) = Θ(n log n). Therefore k should be chosen to be less than or equal to Θ(log n) or $\log_2 n$. Example: if n = 100000 → k = $\log_2 100000 \approx 16$

Problem 4.2: Recurrences

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 ≤ 2.

a) T (n) = 36T (n/6) + 2n

- Master method can be used:

$a = 36$
$b = 6 \quad \rightarrow n^{\log_6 36} = n^2$
$f(n) = 2n \quad \rightarrow f(n) = O(n^{\log_6 36 - \varepsilon})$ *where* $\varepsilon = \log_6 36 - 1 = 2 - 1 = 1$

⇨ **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) + 17$n^{1.2}$

- Master method can be used:

$a = 5$
$b = 3 \quad \rightarrow n^{\log_3 5}$
$f(n) = 17n^{1.2} \rightarrow f(n) = O(n^{\log_3 5 - \varepsilon})$ *where* $\varepsilon = \log_3 5 - 1.2 = 1.465 - 1.2 = 0.264$

⇨ **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 ln(n)

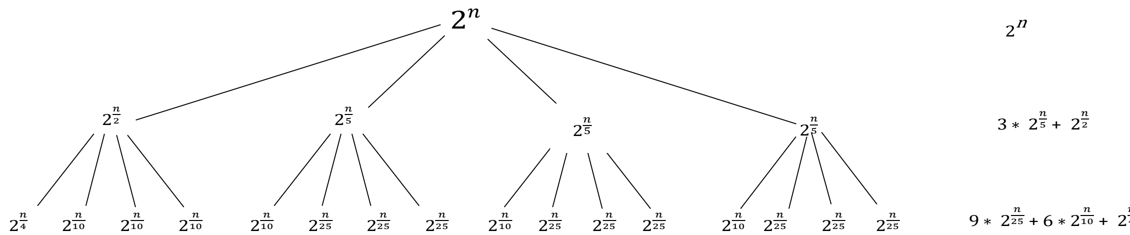
- Master method can be used:

$a = 12$
$b = 2 \quad \rightarrow n^{\log_2 12}$
$f(n) = n^2 \ln(n) \rightarrow f(n) = O(n^{\log_2 12 - \varepsilon})$ *where* $\varepsilon \in \mathbb{R}$

⇨ **Case 1:** f(n) is asymptotically smaller and polynomially smaller than $n^{\log_2 12}$, because $n^{3.585} > n^2$ and Θ(n) ≫ Θ(log n)
Result: $T(n) = \Theta(n^{\log_2 12}) \approx \Theta(n^{3.58})$

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

- Recursion tree method can be used:

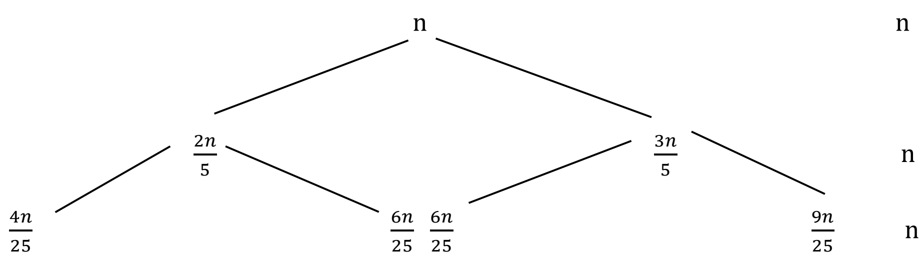


⇨ $2^n\left(1 + \frac{3}{2^{\frac{4n}{5}}} + \frac{1}{2^{\frac{n}{2}}} + \frac{9}{2^{\frac{24n}{25}}} + \frac{6}{2^{\frac{9n}{10}}} + \frac{1}{2^{\frac{3n}{4}}} + \ldots\right)$

⇨ $T(n) = \Theta(2^n)$

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

- Recursion tree method can be used:



⇨ n * h, where h is the height of the tree

! $h = \log n$

⇨ $T(n) = \Theta(n \log n)$