CS/ECE 374 Spring 2025

Heng An Cheng (hacheng2@illinois.edu)

Homework 6 Problem 1

Solution:

(a) We use the function `Merge_Sort_and_Count(1, n, A)` to count the number of inversions in an array A.

The algorithm maintains the following properties:

- The left subarray (indices i to mid) is sorted in increasing order in A.
- The right subarray (indices $mid + 1$ to j) is sorted in increasing order in A.

To count significant inversions efficiently, we iterate over each $A[k]$ in the left subarray and maintain a pointer r in the right subarray such that:

$$A[k] > 10 * A[r].$$

Since both subarrays are sorted, once we find a position r where the condition fails ($A[k] \leq 10 * A[r]$), all subsequent elements in A will also be greater than $10 * A[r]$, leading to an efficient counting process.

Thus, for each $A[k]$, the number of valid r values is $r - (mid + 1)$, contributing to the count of significant inversions.

Algorithm 1 Merge Sort and Count Significant Inversions

```
function MERGE_SORT_AND_COUNT(i, j, A)
    if i ≥ j then
        return 0
    end if
    mid ← ⌊(i + j)/2⌋
    cnt ← 0
    cnt ← cnt + MERGE_SORT_AND_COUNT(i, mid, A)
    cnt ← cnt + MERGE_SORT_AND_COUNT(mid + 1, j, A)
    cnt ← cnt + MERGE_AND_COUNT(i, j, A)
    MERGE(i, j, A)
    return cnt
end function
```

Algorithm 2 Merge and Count Significant Inversions

```
function Merge_And_Count(i, j, A)
    mid ← ⌊(i + j)/2⌋
    l ← i, r ← mid + 1
    cnt ← 0
    for k ← i to mid do
        while r ≤ j and A[k] > 10 × A[r] do
            r ← r + 1
        end while
        cnt ← cnt + (r − (mid + 1))
    end for
    return cnt
end function
```

Algorithm 3 Merge Two Sorted Halves

```
function Merge(i, j, A)
    mid ← ⌊(i + j)/2⌋
    l ← i, r ← mid + 1, id ← i
    while l ≤ mid and r ≤ j do
        if A[l] > A[r] then
            temp[id] ← A[r]
            r ← r + 1
        else
            temp[id] ← A[l]
            l ← l + 1
        end if
        id ← id + 1
    end while
    while l ≤ mid do
        temp[id] ← A[l]
        l ← l + 1
        id ← id + 1
    end while
    while r ≤ j do
        temp[id] ← A[r]
        r ← r + 1
        id ← id + 1
    end while
    for k ← i to j do
        A[k] ← temp[k]
    end for
end function
```

Time Complexity Analysis

Recursive Structure

The algorithm follows a standard merge sort recursion:

$$T(n) = 2T(n/2) + O(n).$$

Thus, we get:

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

Counting Step Complexity

Each element in the left subarray is compared with at most $O(n)$ elements in the right subarray. Since we use a two-pointer technique, we traverse the right subarray at most once per left element, leading to $O(n)$ comparisons in total per merge step.

Merging Step Complexity

Merging two sorted halves takes $O(n)$, as we iterate through both subarrays once.

Overall Complexity

Since the counting and merging steps take $O(n)$ time at each level of recursion, and there are $O(\log n)$ levels, the total complexity remains:

$$O(n \log n).$$

(b) We use a similar approach as in part (a), but introduce a new array B, where:

$$B[i] = W[i] \times A[i].$$

To count the number of significant inversions, we use the function:

`New_Merge_Sort_and_Count(1, n, A, B).`

The algorithm maintains the following properties:

- The left subarray (indices i to mid) is sorted in increasing order in A.
- The right subarray (indices $mid + 1$ to j) is sorted in increasing order in B.

To count significant inversions efficiently, we iterate over each $A[k]$ in the left subarray and maintain a pointer r in the right subarray such that:

$$A[k] > B[r].$$

Since both subarrays are sorted, once we find a position r where the condition fails ($A[k] \leq B[r]$), all subsequent elements in A will also be greater than $B[r]$, leading to an efficient counting process.

Thus, for each $A[k]$, the number of valid r values is $r - (mid + 1)$, contributing to the count of significant inversions.

Algorithm 4 New_Merge_Sort_And_Count(i, j, A, B)

```
function NEW_MERGE_SORT_AND_COUNT(i, j, A, B)
    if i ≥ j then
        return 0
    end if
    mid ← ⌊(i + j)/2⌋
    cnt ← 0
    cnt ← cnt+ NEW_MERGE_SORT_AND_COUNT(i, mid, A, B)
    cnt ← cnt+ NEW_MERGE_SORT_AND_COUNT(mid+1, j, A, B)
    cnt ← cnt+ NEW_MERGE_AND_COUNT(i, j, A, B)
    MERGE(i, j, A)                                ▷ Merge sorted halves in A
    MERGE(i, j, B)                                ▷ Merge sorted halves in B
    return cnt
end function
```

Algorithm 5 New_Merge_And_Count(i, j, A, B)

```
function NEW_MERGE_AND_COUNT(i, j, A, B)
    mid ← ⌊(i + j)/2⌋
    cnt ← 0
    for k ← i to mid do
        r ← mid + 1
        while r ≤ j and A[k] > B[r] do
            r ← r + 1
        end while
        cnt ← cnt + (r − (mid + 1))
    end for
    return cnt
end function
```

Time Complexity Analysis

The time complexity of most of the functions remains the same as in part (a). The only slight difference is that we perform the `Merge` operation twice—once for array A and once for array B. However, this additional merge operation still maintains the same overall time complexity.

At each level of recursion:

- The counting step efficiently tracks significant inversions using a two-pointer technique, which takes $O(n)$.

- The merging step, performed separately for both A and B, also takes $O(n)$.

Since there are $O(\log n)$ levels of recursion (as the problem follows the divide-and-conquer paradigm), the total time complexity remains:

$$O(n \log n).$$

■

CS/ECE 374 Spring 2025 Heng An Cheng (hacheng2@illinois.edu)
Homework 6 Problem 2

Solution:

(a) Define $dp[k]$ as the maximum score achievable before dancing to the k-th song.

Base Case

- $dp[0] = 0$ because no dances have been performed yet.
- $dp[1] = 0$ since we haven't chosen a dance before the first song.

Transition

At each song k, we have two choices:

(a) **Skip song k:** The best score remains the same as the previous song:

$$dp[k] = \max(dp[k-1], dp[k])$$

(b) **Dance to song k:** If we choose to dance, we gain $\texttt{Score}[k]$, but we must rest for $\texttt{Wait}[k]$ songs. The next song available for dancing is $k + \texttt{Wait}[k] + 1$. We update $dp[k + \texttt{Wait}[k] + 1]$ to store the maximum score achieved up to this new available song:

$$dp[k + \texttt{Wait}[k] + 1] = \max(dp[k] + \texttt{Score}[k], dp[k + \texttt{Wait}[k] + 1])$$

If dancing to k means we reach beyond n, we update a separate variable `max_score` to track the best possible score obtained.

Final Computation

Since some values may be stored in `max_score`, we take the maximum of `max_score` and $dp[n]$ before returning the result.

Time Complexity Analysis

The algorithm processes each song exactly once, iterating through $k = 1$ to n. Within each iteration, we perform only constant-time operations (comparison and update of dp).

Algorithm 6 Compute Maximum Dance Score

```
Input: Arrays Score[1..n] and Wait[1..n]
Output: Maximum achievable dance score
dp[0] ← 0
dp[1] ← 0
max_score ← 0
for k ← 1 to n do
    dp[k] ← max(dp[k-1], dp[k])
    if k + Wait[k] + 1 ≤ n then
        dp[ k + Wait[k] + 1 ] ← max(dp[k] + Score[k], dp[k + Wait[k] + 1])
    else
        max_score ← max(dp[k] + Score[k], max_score)
    end if
end for
max_score ← max(max_score, dp[n])
return max_score
```

■