Design and Analysis of Algorithms Project

T25 and P11

Arya[1] Madhubala[2]

19 October 2025

Table of Contents

Problem Statement

Problem

Counting smaller elements after self.

The objective is to take an array as input and output another array. The i'th element as output is the number of numbers in the array smaller than the i'th element in the input. We do not look at the elements before the i'th element which are smaller.
Eg. Input = [7, 1, 3, 2]
Output = [3, 0, 1, 0].

Psuedocode

function merge(a,l,m,r,count)

n1 $\leftarrow m - l + 1$
n2 $\leftarrow r - m$
L $\leftarrow$ copy of $a[l, l+1, .., m]$ Left subarray
R $\leftarrow$ copy of $a[m, m+1, .., r]$ Right subarray
i $\leftarrow 0$
j $\leftarrow 0$
k $\leftarrow l$
rightcount $\leftarrow 0$ // For # right-side elements merged before this
while $i < n1$ and $j < n2$
if i'th element of L <= j'th element of R
count[L[i].index] $\leftarrow count[L[i].index] + rightCount$ // Add the smaller elements to it's count.
a[k] $\leftarrow L[i]$
i $\leftarrow i + 1$ $k \leftarrow k + 1$

Psuedocode Continued

merge continued

```
  else
    rightCount ← rightCount + 1;
     a[k] ← R[j]
     j ← j + 1
     k ← k + 1
end while
while i ≤ n1 :
  a[k] = L[i]
  count[L[i].index] = count[L[i].index] + rightcount
  i ← i + 1
  k ← k + 1
end while
while j ≤ n2 :
  a[k] = R[j]    j ← j + 1
```

Pseudocode Cotinued

merge continued

k $\leftarrow k + 1$

function mrgsrt(a, l, r, count)

if l >= r then
  return
m $\leftarrow l + (r - l)/2$
mrgsrt(a, l, m, count)
mrgsrt(a, m + 1, r, count)
merge(a, l, m, r, count)

function main

arr $\leftarrow$ Input
n $\leftarrow$ length(arr)
a $\leftarrow$ empty list
f or i

Pseudocode continued

main continued

```
from 0 to n-1
  append (arr[i], i) to a // store value with original index
 count ← array of zeros of size n
 call mrgsrt(a, 0, n-1, count)
 print count
```

Approach & Method

Simply it is clear that element wise comparison is needed. Now the task was to minimize the number of comparisons done. So the basic sorting technique merge sort which deals with breaking the array in log n time and merging them back. After the basic comparing each element with itself which would have taken $O(n^2)$ this approach seemed a faster and better way to do it.

Hence the method used is simple Merge Sort. The explanation provided is in terms of how the flow of the code goes rather than the order of the pseudo-code.

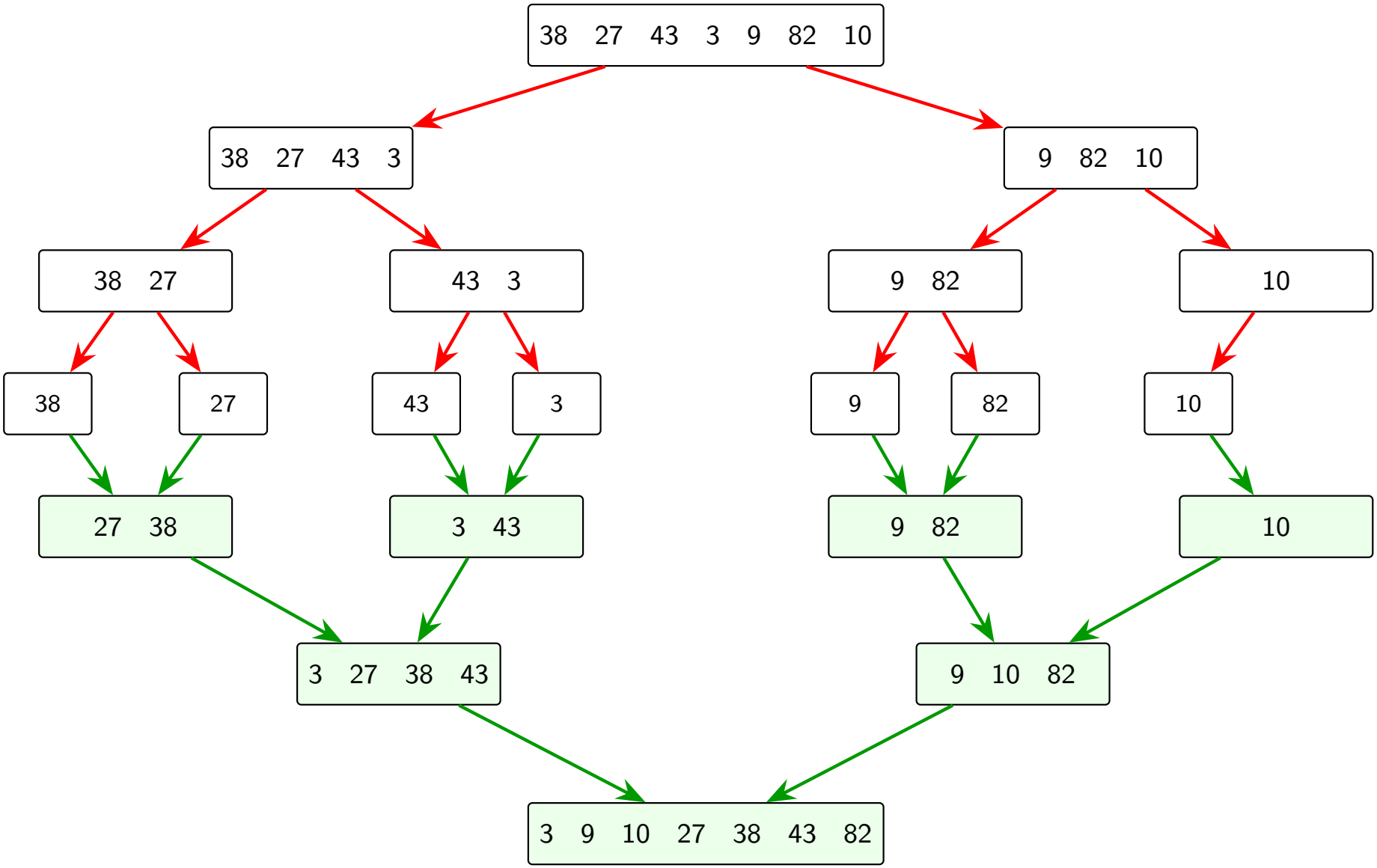
We sub-divide our array into smaller and smaller elements so that we can finally compare them. Now the sub-division is into left and right arrays till they are single elements then we stop the recursion. Then we merge the elements after certain operations to get a sorted array. Now we create two array's called the left array and right array, using these we check if a left array elements is lesser than a right array elements.

Approach & Method Continued

If yes,then we can just add the number of elements to our count (output) array that already have been sorted. If there is such an element then only the count value will increase otherwise it implies that value is the smallest in the array. Now otherwise if a left array element is greater than a right array element then clearly it needs to go in front of the current element in the left array. Hence like this we keep updating our count array and we display it as our result.

If yes,then we can just add the number of elements to our count(output) array that already have been sorted.If there is such an element then only the count value will increase otherwise it implies that value is the smallest in the array

Merge Sort Visualization



Complexity Analysis

The algorithm is based on the divide-and-conquer technique of Merge Sort. We recursively divide the array into two halves until each subarray contains a single element.

During the merge step, the elements from the left and right subarrays are compared, and for each element, the number of smaller elements appearing after it (i.e., in the right subarray) is updated in the count array.

Since each merge operation processes every element once, the time taken for each merge step is $\Theta(n)$.

The recursive division produces two levels (right and left), hence giving us $T(n/2) + T(n/2)$ giving a recurrence relation $\mathbf{T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)}$.

Using Master Theorem we get the time complexity of $\mathbf{\Theta(n \log n)}$.

Complexity Analysis Continued

The auxiliary space used includes temporary arrays during each merge step which contribute to $O(n)$ space and a recursion stack of depth $\log n$, resulting in a space of $O(\log(n))$.

Overall space complexity is $\mathbf{O(n)}$ since $\log(n)$ is dominated by n.

The algorithm maintains the same asymptotic complexity for best, average, and worst cases, making it efficient for counting smaller elements after each element in the array.

Correctness

Correctness Proof

Claim 1

At each merge step, the left and right subarrays are individually sorted before merging.

Proof. We do induction on every recursive step
Base case: If subarray has 1 element then it is already sorted.
Inductive step: Assume recursive calls correctly sort both halves. Then the merge procedure merges two sorted arrays into a single sorted one. We compare each element of the left and right array and put the smallest element first. Like this we ensure a sorted array at every stage.

Correctness Continued

Claim 2

When merging, every time an element from the right half is smaller than an element from the left half, it contributes exactly one "smaller-after" count for that left element.

Proof: Suppose L[i] $\geq$ R[j] during merge. Since both L and R are sorted, all elements from R[j] onward are $\geq$ R[j]. All unmerged elements in the left (L[i], L[i+1], ...) are $\geq$ L[i].

Hence, when we move R[j] before L[i], every remaining element in L[i..end] must be larger than R[j]. Therefore, each of them gains +1 to their "smaller after self" count.

Thus, we increment the count for the left elements appropriately at every such comparison.

Correctness Continued

Claim 3

Each element's count equals the total number of smaller elements to its right in the original array.

Proof by induction on size of subarray:
Base case: Single element array implies count is 0 (no elements to the right).

Inductive step: Suppose the count is correct for both halves after recursive calls. When merging, opposite pairs (one in left, one in right) are handled by Claim 2. Since both halves are already correctly counted and we now add exactly the opposite-pair contributions, the final count for each element matches.

Correctness Continued

Claim 4

All elements and counts remain aligned with their original indices.

The algorithm tracks each element with its index (implicitly or explicitly) through recursive sorting and merging.

During merge, we place back elements in sorted order while updating the count array according to their original indices.

So, now we want to define a loop invariant for this loop something that is true before and after every iteration hence ending our correctness justification.

Loop Invariant: At the start of every iteration of the merge loop:

Correctness Loop Invaraint

1. The portion a[l..k-1] contains the kl smallest elements of L ∪ R, in sorted order.
2. For every element in L[0..i-1], the number of smaller elements to its right (from R) has been correctly counted.
3. The elements in L[i..end] and R[j..end] are still unmerged.

At the start of every iteration of the while loop we try to ensure The portion a[l..k-1] contains the $k - l$ smallest elements of L U R, in sorted

1 Order.

2 For every element in L[0..i-1], the number of smaller elements to its right (from R) has been correctly counted.

3 The elements in L[i..end] and R[j..end] are still not merged.

Proof 1

Before the first iteration: k = l, i = 0, j = 0. a[l,l+1,...k-1]=a[0] is empty hence trivially sorted. Count is still 0 at all entries since while statement hasn't been executed yet. Both sub-arrays L and R remain same. So the invariant holds before the loop starts i.e for the base case

Proof 2

The invariant holds at the start of the loop. We show it remains true after one iteration.

1. L[i] $\leq$ R[j] Then we put L[i] into a[k] (i.e a[0]) Since R[j] and all later R elements R[k] $\geq$ R[j] $\geq$ L[i] for all k $\geqslant$ j, we know a[l,...,k] remains sorted.

   The count for L[i] is unchanged (since we did not find any smaller element). We increment i and k. Invariant still holds.

2. L[i] $\geqslant$ R[j] Then we put R[j] into a[k]. Every unmerged L[i,...,end] is larger than R[j], which means R[j] is smaller than each L[i]. So for each of those L elements, their "smaller-after" count must increase by 1(hence updating count). a[l,...,k] remains sorted since we placed the smallest remaining element next. Invariant still holds.

Correctness Continued

3 When the loop ends Either i = l1 or j = r1. The invariant ensures a[l,...,k-1] is sorted and all smaller counts for merged elements are correct. The remaining elements (if any) in L or R are just added since it's already sorted and no new smaller elements remain to be counted. So upon termination, the entire merged range a[l..r] is sorted, and all "smaller-after" counts are correct.

Hence this ends our correctness argument.

Experiments & Datasets

Algorithm 1 Job Sequencing with Deadlines (Greedy)

Input: Array of jobs $J = \{(id_i, deadline_i, profit_i)\}_{i=1}^{n}$
Output: Maximum total profit with sequence of jobs.
procedure MERGESORT(a, l, r)
  if $l < r$ **then**
    $m \leftarrow \lfloor (l + r)/2 \rfloor$
    MERGESORT(a, l, m)
    MERGESORT$(a, m + 1, r)$
    MERGE(a, l, m, r)
  end if
end procedure
procedure MERGE(a, l, m, r)
  Copy left half $L = a[l, ..., m]$, right half $R = A[m + 1..r]$
  $i \leftarrow 0$, $j \leftarrow 0$, $k \leftarrow l$
  while $i < |L|$ and $j < |R|$ **do**
    if $L[i].profit \geq R[j].profit$ **then**
      $A[k] \leftarrow L[i]$; $i \leftarrow i + 1$
    else
      $A[k] \leftarrow R[j]$; $j \leftarrow j + 1$
    end if
    $k \leftarrow k + 1$
  end while
  while $i < |L|$ **do**
    $A[k] \leftarrow L[i]$; $i \leftarrow i + 1$; $k \leftarrow k + 1$
  end while
  while $j < |R|$ **do**
    $A[k] \leftarrow R[j]$; $j \leftarrow j + 1$; $k \leftarrow k + 1$
  end while
end procedure
procedure SCHEDULEJOB(J)
  ▷ Step 1: Sort jobs by profit (descending)
  MERGESORT$(J, 0, n - 1)$
  ▷ Step 2: Find maximum deadline
  $maxDeadline \leftarrow \max_i(deadline_i)$
  ▷ Step 3: Initialize empty time slots
  $slot[1, ...maxDeadline] \leftarrow -1$
  $totalProfit \leftarrow 0$
  ▷ Step 4: Greedy assignment
  for each job j in J (in sorted order) **do**
    for $t \leftarrow j.deadline$ **to** 1 **do**
      if $slot[t] == -1$ **then**
        $slot[t] \leftarrow j.id$
        $totalProfit \leftarrow totalProfit + j.profit$
        break
      end if
    end for
  end for
  ▷ Step 5: Output scheduled jobs and profit
  Print all $slot[t] \neq -1$

Experiments & Datasets Continued

Above the greedy algorithm for problem maximizing profit with given deadlines. Now that's the algorithm. For the correctness we need to ensure that after t steps we have the t most profitable jobs in our first t places of our schedule jobs array.
When the algorithm considers a new job j (with maximum remaining profit), it checks from $t = j.deadline$ downwards. If a slot is free, it assigns j to the latest possible time. This does not hamper previous jobs, because earlier jobs already occupy earlier or later distinct slots, and the current assignment respects deadlines.
Hence, after placing j, the invariant holds. When all jobs are processed, the invariant ensures that:

All occupied slots correspond to the most profitable feasible subset of jobs.

Therefore, the algorithm outputs an optimal schedule.

Complexity Time and Space

$$
\begin{aligned}
T(n) &= O(n \log n) && \text{(merge sort)} \\
&\quad + O(n \cdot d_{\max}) && \text{(slot assignment)} \\
&= O(n \log n + n d_{\max})
\end{aligned}
$$

Space Complexity: $O(n + d_{\max})$

Results

Input: (1, 2, 100), (3, 2, 27), (4, 1, 25), (2, 1, 19), (5, 3, 15)
Output: Scheduled Jobs: J3 J1 J5. Total Profit: 142
Input:
J1 $\rightarrow$ deadline: 5, profit: 82
J2 $\rightarrow$ deadline: 17, profit: 45
J3 $\rightarrow$ deadline: 8, profit: 91
J4 $\rightarrow$ deadline: 2, profit: 15
J5 $\rightarrow$ deadline: 13, profit: 74
J6 $\rightarrow$ deadline: 1, profit: 59
J7 $\rightarrow$ deadline: 20, profit: 98
Output:
Scheduled Jobs: J7 J3 J1 J5 J6. Total Profit: 404

Results

Input:
(1, 5, 200),
(2, 3, 180),
(3, 3, 190),
(4, 2, 300),
(5, 4, 120),
(6, 2, 100),
(7, 1, 250)
Output: Scheduled Jobs: J4 J3 J1 J2 J5. Total Profit: 990
Input:
(1, 4, 150), // Machine part assembly
(2, 2, 200), // Gear production
(3, 3, 180), // Motor coil winding
(4, 1, 120), // Quick prototype order
(5, 5, 100) // Packaging line test
Output: Jobs scheduled: J2 J3 J1 J5. Total Profit: 630

Results

Input:
(1, 20, 900), (2, 18, 850), (3, 17, 800), (4, 16, 750), (5, 15, 700), (6, 14, 650), (7, 13, 600), (8, 12, 580), (9, 11, 550), (10, 10, 500), (11, 9, 480), (12, 8, 450), (13, 7, 420), (14, 6, 400), (15, 5, 370), (16, 4, 340), (17, 3, 310), (18, 2, 290), (19, 1, 260), (20, 19, 870)
Output: Scheduled Jobs: J19 J18 J17 J16 J15 J14 J13 J12 J11 J10 J9 J8 J7 J6 J5 J4 J3 J2 J20 J1 Total Profit: 11070

All the time analysis is attached to the github since overleaf was exceeding time computations.

Challenges

1. Time Complexity Issues

The O(n $\times$ m) scheduling phase runs into serious problems under certain conditions:

- When the max deadline gets very large (say, m = 10,000), the algorithm struggles
- Things get worse when multiple jobs are competing for similar time slots
- Consider this: scheduling 1000 jobs with a deadline of 1000 means roughly 1 million iterations

2. Space Inefficiency

The slot array size equals the maximum deadline, which creates a major problem. If just one job has a deadline of 1,000,000 while all others have a deadline of 10, you're still allocating 1 million slots. This wastes a lot of memory, especially when deadlines are sparsely distributed.

Challenges Continued

3. Fixed Time Slots

The current approach assumes unit-time jobs—meaning each job takes exactly one time unit. This is a significant constraint because it can't handle jobs with varying durations.

4. No Tie-Breaking Strategy

When two jobs have equal profits, the order becomes arbitrary. In practice, you might want to prioritize by deadline or job ID to get deterministic, reproducible results.

Challenges Continued

5. Memory Allocation in Merge Sort

At every merge step, the algorithm creates temporary vectors L and R. With really large datasets, this can lead to performance bottlenecks since you're constantly allocating and deallocating memory.

6. Limited Output Information

There are a few gaps in what the algorithm tells you:

- It doesn't show which jobs were rejected
- There's no information about slot utilization percentage
- You can't see individual job assignments to specific time slots

7. No Error Handling

The implementation is missing some basic safeguards:

- Doesn't validate input for things like negative profits or zero deadlines
- No bounds checking

Scalability

Performance Comparison

Approach	Time	Space	Best For
Original	$O(n \log n + nm)$	$O(n + m)$	Small m
DSU	$O(n \log n)$	$O(n + m)$	General case
Set-based	$O(n \log m)$	$O(n + k)$	Sparse deadlines
Parallel	$O(n \log n/p)$	$O(n + m)$	Multi-core systems

Where: $n =$ jobs, $m =$ max deadline, $k =$ unique slots used, $p =$ processors

For most cases, use DSU approach — it provides the best balance of:

1. Optimal time complexity
2. Simple implementation
3. Scalable to large inputs
4. Maintains correctness

Github

The github link is here: https://github.com/Crysis29/DAA-Project-?tab=readme-ov-file#readme

Thank You

Team - Die Tum Tum

Arya and Madhubala