ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Final Exam, Algorithms 2017-2018

- You are only allowed to have a handwritten A4 page written on both sides.
- Communication, calculators, cell phones, computers, etc... are not allowed.
- Your explanations should be clear enough and in sufficient detail that a fellow student can understand them. In particular, do not only give pseudocode without explanations. A good guideline is that a description of an algorithm should be such that a fellow student can easily implement the algorithm following the description.
- Attached at the end of the exam is a French translation.
- **Do not touch until the start of the exam.**

Good luck!

Name: _____ **N° Sciper:** _____

Problem 1 / 23 points	Problem 2 / 14 points	Problem 3 / 16 points	Problem 4 / 17 points	Problem 5 / 18 points	Problem 6 / 12 points

Total / 100

1 *(23 pts)* **Basic questions.** This problem consists of five subproblems (1a-1e) for which you **do not** need to motivate your answers.

1a *(5 pts)* Arrange the following functions in increasing order according to asymptotic growth.

$$n^{\sqrt{n}},\ \log n,\ 3^n,\ n(\log n)^5,\ n^4 \log n,\ \log(n!)$$

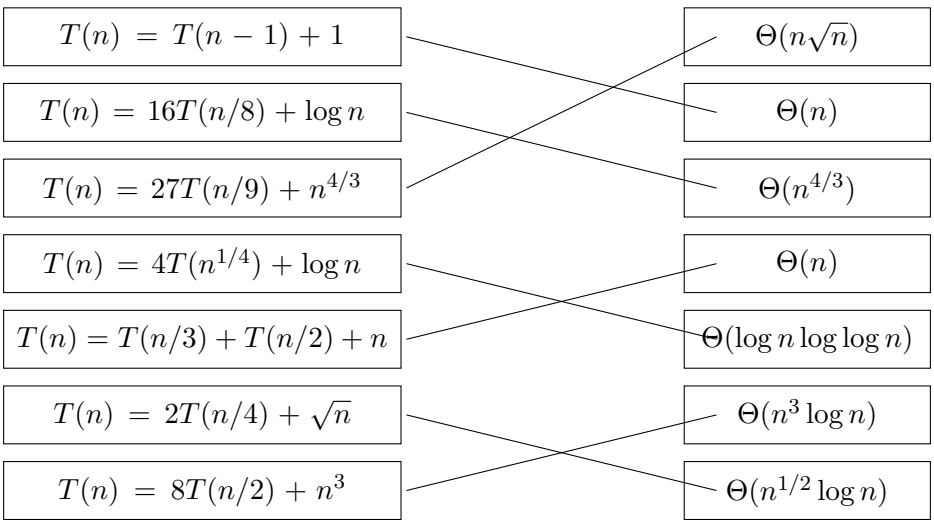
(In this problem, you only need to give the right order, i.e., you do not need to explain your answer.)

Solution:

$$\log n,\ \log(n!),\ n(\log n)^5,\ n^4 \log n,\ n^{\sqrt{n}},\ 3^n$$

1b *(6 pts, 1 pt for each correct pair)* Pair up each of the recurrence relations on the left side with the correct asymptotic growth function on the right side (assuming that $T(1) = 1$). For example the recurrence $T(n) = T(n-1) + 1$ has asymptotic growth $\Theta(n)$.

Solution:



1c *(4 pts)* Consider the recurrence $T(n) = T(\alpha n) + T(\beta n) + n^2$, $T(1) = \Theta(1)$, for some constants $\alpha, \beta \in (0, 1)$ that satisfy $\alpha^2 + \beta^2 = 1$. Give a tight asymptotic bound for $T(n)$. You do not need to motivate your answer.

Solution:

The answer is $T(n) = \Theta(n^2 \log n)$

1d *(4 pts)* Consider the recurrence $T(n) = T(\alpha n) + T(\beta n) + n^2$, $T(1) = \Theta(1)$, for some constants $\alpha, \beta \in (0, 1)$ that satisfy $\alpha^2 + \beta^2 < 1$. Give a tight asymptotic bound for $T(n)$. You do not need to motivate your answer.

Solution:

The answer is $T(n) = \Theta(n^2)$

1e *(4pts)* Suppose that you want to store n keys in a hash table with m slots. Assume simple uniform hashing, and define a 4-collision as a set of four **distinct** keys $\{k_1, k_2, k_3, k_4\}$ that are hashed into the same slot.

Give a tight asymptotic expression for the expected number of 4-collisions.

Solution:

The expected number of 4-collisions is $\Theta(n^4/m^3)$

2 *(14 pts)* **Maintaining the second smallest element.** In this question you should design and analyze a data structure that supports the following operations:

- INSERT(x): add an integer x to the data structure
- FINDSECONDSMALLEST(): return the second smallest element if the data structure contains more than one element and NULL otherwise
- DELETESECONDSMALLEST(): remove the second smallest element (if the data structure contains more than one element)

For full credit the operations must have the following time complexities: $O(\log n)$ for INSERT, $O(1)$ for FINDSECONDSMALLEST, $O(\log n)$ for DELETESECONDSMALLEST.

In this problem you are required to (a) describe your data structure and (b) analyze runtime of the three operations above.

Solution:

We present three solutions: two based on a heap and one that can be based on either a heap or a number of other data structures.

Solution 1: heap and the min element on the side. Here, our data structure is just a standard min-heap H (i.e., a max-heap with the order of elements reversed) and a single variable m. The intention is that m shall hold the minimum element (whenever the structure is non-empty), and H shall hold all the remaining elements. We begin with H empty and $m = \infty$. We discuss how the three operations can be implemented:

- INSERT(x): compare x to m. If $m < x$, then m should remain the smallest element, and we insert x into H. Otherwise m should be replaced by x as the smallest element, and so we insert m into H (unless $m = \infty$, in which case do nothing) and then set $m := x$.
- FINDSECONDSMALLEST(): if H is non-empty, return the top element of H. Otherwise return nil.
- DELETESECONDSMALLEST(): if H is non-empty, run DELETEMIN on H. Otherwise do nothing.

The three operations clearly have the same asymptotic runtime as the corresponding heap operations, which is $O(\log n)$, $O(1)$, and $O(\log n)$ respectively.

Solution 2: just a heap. Here, all the elements are kept in a standard min-heap.

- INSERT(x): just insert x into the heap.
- FINDSECONDSMALLEST(): note that, by the heap property, the second-smallest element in a heap is one of the two children of the root. We return the smaller of the two. (If there are fewer than two elements, return nil.)
- DELETESECONDSMALLEST(): we start by locating the second-smallest element, as above. Then we delete it and restore the heap property manually. For instance, we can assign the value ∞ to the deleted element, push it down to a leaf (always pushing to the subtree containing the smaller element), then delete it.

We might also consider a solution that keeps only a heap and, for the second and third operation, first removes the largest element, storing it on the side, then performs the wanted operation on the heap so obtained, and then adds the deleted element back. However, such a structure would have a running time of $O(\log n)$ for FindSecondSmallest(), whereas we need $O(1)$.

Solution 3: heap or a balanced binary tree, and the second-smallest element on the side. Finally, we can use any data structure that supports operations Insert, FindSmallest and DeleteSmallest in $O(\log n)$ time. (Note that we do not require $O(1)$ time for FindSmallest here; we will be fine with $O(\log n)$.) We will also keep the second-smallest "on the side", as a variable m_2 (initially nil), ready to be returned. Note that, differently from Solution 1, m_2 will also be present in the heap/tree.

- Insert(x): just insert x into the heap/tree. Then update m_2 by finding the second-smallest element in the heap/tree. (For example, in a heap, it is one of the two children of the root; in a balanced binary tree, it can be found by first finding the smallest element, and then its successor. Alternatively, regardless of the data structure used, we can just: delete the smallest element, storing it on the side as m, find the now-smallest element and save it as m_2, and then insert back the deleted smallest element m. The entire step takes $O(\log n)$ time in any case.)
- FindSecondSmallest(): just return m_2! (This step takes $O(1)$ time. By storing m_2, we do not need the underlying data structure to have $O(1)$ runtime for FindSmallest nor DeleteSmallest.)
- DeleteSecondSmallest(): again, we can either perform the operation in some way depending on the underlying data structure, or use the heap/tree as a black-box: first delete the smallest element and save it on the side as m, then delete the now-smallest element, then save the now-smallest element as m_2, and then add back the deleted element m. (This takes $O(\log n)$ time.)

3 *(16 pts)* **String similarity.** In this problem you will design an algorithm that, given two strings s and t, outputs the length of the **longest common substring** between s and t.

- **Input:** The input consists of two strings s and t of length n and m respectively.
- **Output:** The length of the longest common substring: the largest k such that there exist $1 \le i \le n-k+1, 1 \le j \le m-k+1$ such that $s[i:i+k-1] = t[j:j+k-1]$. Here $s[a:b]$ stands for the substring of s corresponding to indices $a, a+1, \ldots, b$.

For example, the maximum common substring between $s =$ `ABAABABBAB` and $t =$ `AAABBAA` is `ABBA`, of length 4. Indeed, $s[6:9] = t[3:6] =$ `ABBA`, and there is no common substring of length larger than 4.

Give a dynamic programming solution to the problem with runtime $O(nm)$. **Note: this is similar, but not the same as the longest common subsequence problem covered in class.**

In this problem you are required to (a) define the subproblems and the recurrence, (b) analyze the runtime of a bottom-up implementation of your recurrence.

Solution: Let $d(i,j)$ be the length of the longest common substring that finishes at $s[i]$ and $t[j]$ for all $0 \le i \le n, 0 \le j \le m$. If $s[i] \ne t[j]$, then there is no common substring that finishes at this position, therefore the answer is zero. On the other hand, if $s[i] = t[j]$ then the length of such string is one more than the string that finishes at position $s[i-1]$ and $t[j-1]$. We thus get that $d(i,j)$ satisfies the following recurrence:

$$d(i,j) = \begin{cases} 0 & i = 0 \text{ or } j = 0 \text{ or } s[i] \ne t[j] \\ d(i-1, j-1) + 1 & \text{otherwise} \end{cases}$$

Then the answer is the maximum over all $0 \le i \le n, 0 \le j \le m$, of $d(i,j)$.

It is clear that the running time of filling this table is $\Theta(mn)$, since our table has $\Theta(mn)$ many cells and filling each takes $O(1)$. Therefore we can find the solution by finding the longest number in the table, which is also $\Theta(mn)$. A bottom-up implementation of our algorithm is given below as Algorithm 1:

Algorithm 1 Longest common substring

```
1:  function LongestCommonSubstring(s, t, n, m)
2:      Initialize d[0..n][0..m] to zero
3:      answer ← 0
4:      for i=1 do to n
5:          for j=1 do to m
6:              if s[i]==t[j] then
7:                  d[i][j] = d[i − 1][j − 1] + 1
8:                  answer= max(answer, d[i][j])
9:              end if
10:         end for
11:     end for
12:     return answer
13: end function
```

4 *(17 pts)* **Algorithms on a grid.** You are given an $n \times n$ grid of cells, with every grid cell containing one of the letters `A`, `L`, `G`, `O`. Your task is to find as many non-overlapping spellings of the word `ALGO` on the grid as possible (see Fig. 1). Formally, a spelling of `ALGO` on the grid is a sequence of grid cells $(i_1, j_1), (i_2, j_2), (i_3, j_3), (i_4, j_4) \in \{1, 2, \ldots, n\} \times \{1, 2, \ldots, n\}$ such that the letter in position (i_1, j_1) is `A`, the letter in position (i_2, j_2) is `L`, the letter in position (i_3, j_3) is `G`, the letter in position (i_4, j_4) is `O`, and consecutive cells are adjacent.

We say that two cells (i, j) and (i', j') are adjacent on the grid if either $i = i'$ and $|j - j'| = 1$ or $|i - i'| = 1$ and $j = j'$. Two spellings are called non-overlapping if they do not share cells.

Input: An $n \times n$ array A with $A_{ij} \in$ `{A, L, G, O}` for every $1 \le i \le n, 1 \le j \le n$.

Output: The largest $k \ge 0$ such that there exist k **non-overlapping** spellings of `ALGO` on the grid.

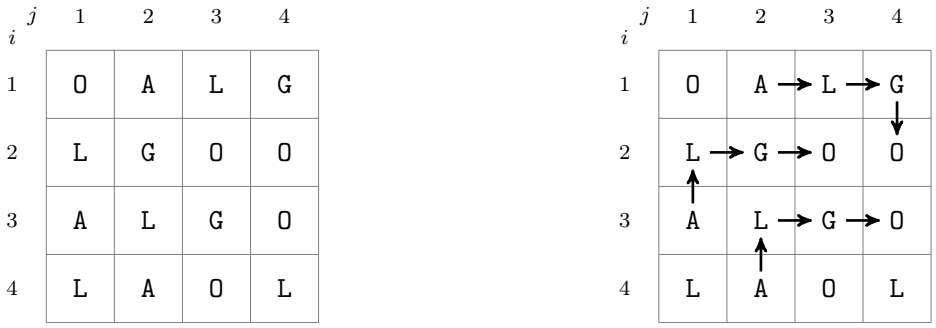


Figure 1. 4×4 instance of the problem (left) and the optimal solution (right).

Design and analyze an algorithm for the problem. Your algorithm should run in time polynomial in n. *Hint: construct a graph G with a source s and a sink t such that the value of max-flow from s and t in G is exactly the maximum number of non-overlapping spellings.*

In this problem you are required to (a) describe an efficient algorithm and (b) analyze its runtime.

Solution: This problem can be solved by associating transforming the input instance into a graph and finding the maxflow on it. We construct the graph $G(V, E)$ as follows.

The set of the vertices V is defined as $V = \left\{v_i^j;\ \text{for all } i, j \in [n]\right\} \cup \{s\} \cup \{t\}$. Each v_i^j corresponds to a cell of the grid. The set of edges is defined as follows. For all $i, j \in [n]$

$$(v_i^j, v_i^{j+1}) \in E \text{ iff } (A_{i,j}, A_{i,j+1}) \in \{(\texttt{A}, \texttt{L}), (\texttt{L}, \texttt{G}), (\texttt{G}, \texttt{O})\}$$

$$(v_i^j, v_i^{j-1}) \in E \text{ iff } (A_{i,j}, A_{i,j-1}) \in \{(\texttt{A}, \texttt{L}), (\texttt{L}, \texttt{G}), (\texttt{G}, \texttt{O})\}$$

$$(v_i^j, v_{i+1}^j) \in E \text{ iff } (A_{i,j}, A_{i+1,j}) \in \{(\texttt{A}, \texttt{L}), (\texttt{L}, \texttt{G}), (\texttt{G}, \texttt{O})\}$$

$$(v_i^j, v_{i-1}^j) \in E \text{ iff } (A_{i,j}, A_{i-1,j}) \in \{(\texttt{A}, \texttt{L}), (\texttt{L}, \texttt{G}), (\texttt{G}, \texttt{O})\}$$

And also we connect s and t to nodes v_i^j's as follows:

$$(s, v_i^j) \in E \text{ iff } A_{i,j} = \texttt{A}$$

$$(v_i^j, t) \in E \text{ iff } A_{i,j} = \texttt{O}$$

We compute the maximum $s - t$ flow in this graph with capacities of all the edges equal to 1 and capacity of all v_i^j vertices equal to 1. The maximum flow on this graph is equal to the maximum number of non overlapping spellings of `ALGO` (see below for correctness proof). The construction of the graph take time linear in the size of the input if we use the adjacency array format to represent the graph. We then find the maximum flow using Ford-Fulkerson algorithm, using BFS to find augmenting paths at every iteration. The number of iterations is upper bounded by the number of spellings, and thus by n^2, and each iteration takes $O(n^2)$ time. Overall, our algorithm runs in $O(n^4)$ time.

To see the correctness, note that because we have the constraint on the maximum capacity of each v_i^j being at most one there will be no overlap between the spellings. First we show that for any set of non overlapping spellings on the grid there exists a valid flow on the graph. The reason is that for each spelling of `ALGO` on the grid there is a sequence of adjacent grid cells $(i_1, j_1), (i_2, j_2), (i_3, j_3), (i_4, j_4) \in \{1, 2, \ldots, n\} \times \{1, 2, \ldots, n\}$ and we can send a unit amount flow from s to $v_{i_1}^{j_1} \to v_{i_2}^{j_2} \to v_{i_3}^{j_3} \to v_{i_4}^{j_4}$ and to t. This does not violate the capacities of the edges or nodes because spellings on the cell were non overlapping.

Now we show that any integral flow on the graph corresponds to a set of valid non overlapping spellings of `ALGO` on the grid (note that it is enough to consider integral flows because Ford-Fulkerson returns an integral flow whenever capacities are integral, as in our case). First clearly the capacity constraints on the nodes prevents the overlaps. Also by the connections from the nodes it is clear that each unit of flow corresponds to a spelling of `ALGO`.

5 *(18 pts)* **Minimum-weight forests.** In this problem you are given a connected undirected graph $G = (V, E)$ with non-negative edge weights w, and your task is to **design and analyze** an efficient algorithm that computes the minimum-weight forest in G with exactly two connected components.

Input: Connected undirected graph $G = (V, E)$ with non-negative edge weights $w : E \to \mathbb{R}$.

Output: Minimum-weight forest in G with exactly two connected components.

For full credit, your algorithm should run in time $O((|V| + |E|) \log |V|)$.

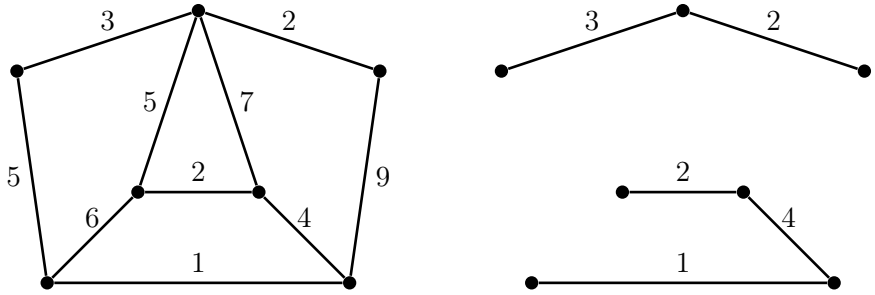


Figure 2. Example of input graph G with edge weights (left), and its minimum-weight forest with two components (right).

In this problem you are expected to (a) design an efficient algorithm (b) prove its correctness and (c) analyze its runtime.

Solution: Run Kruskal's algorithm for $|V| - 2$ iterations, which results in a forest with 2 connected components – denote this forest by F. We show that this forest is the cheapest forest with two components in G. Let T denote the minimum spanning tree resulting from running Kruskal's algorithm to the end, i.e. for $|V| - 1$ iterations. Recall that Kruskal's algorithm adds edges to the minimum spanning tree T in non-decreasing order of weight, so the forest F that we constructed is in fact T without its most expensive edge. To show that F is a minimum-cost forest with 2 connected components, we argue by contradiction. Let F' be a minimum-cost forest with two components in G, and suppose that the cost of F' is strictly smaller than the cost of F. Now there exists at least one edge e' of T which can be added to F' to turn it into a spanning tree T', and the weight of T' is the sum of the weight of F' and the weight of e': $cost(T') = cost(F') + w(e')$. On the other hand, the weight of T is the weight of F plus the weight of a heaviest edge in T: $cost(T) = cost(F) + \max_{e \in T} w_e$. Putting these two bounds together, we get

$$cost(T') = cost(F') + w(e') < cost(F) + w(e') \leq cost(F) + \max_{e \in T} w_e = cost(T),$$

which contradicts the assumption that T was a minimum spanning tree.

6 *(12 pts)* **Route intersections revisited.** The infinite glacier in the Swiss Alps has the following structure: two perfectly parallel walls (North and South) with a river of ice flowing between them. We model the South wall as the line $y = 0$ in the plane, and the North wall as the line $y = 1$ (see Fig. 3). Now n companies want to establish Tyrolean routes across the glacier, and each company designated two climbers, who occupy positions on the two walls and have a rope between them. The ropes may cross, and in this problem your task is to design an efficient algorithm for **counting the number of crossings**.

Input: Positions $a_i, i = 1, \ldots, n$ of n climbers on the North wall. The first climber in the i-th pair occupies position $(a_i, 1)$ on the North wall, and the second occupies position $(i, 0)$ on the South wall (see Fig. 3). The rope between them is the line segment connecting $(a_i, 1)$ to $(i, 0)$ in the plane.

Output: Your task is to compute **the number of intersections** of the line segments.

You may assume that n is a power of two if this simplifies your analysis, as well as that no two climbers occupy the same position, and no three line segments intersect at the same point. You may use the fact that a segment connecting $(i, 0)$ to $(a_i, 1)$ intersects a segment connecting $(j, 0)$ to $(a_j, 1)$ if and only if $i < j$ and $a_i > a_j$ or $i > j$ and $a_i < a_j$.

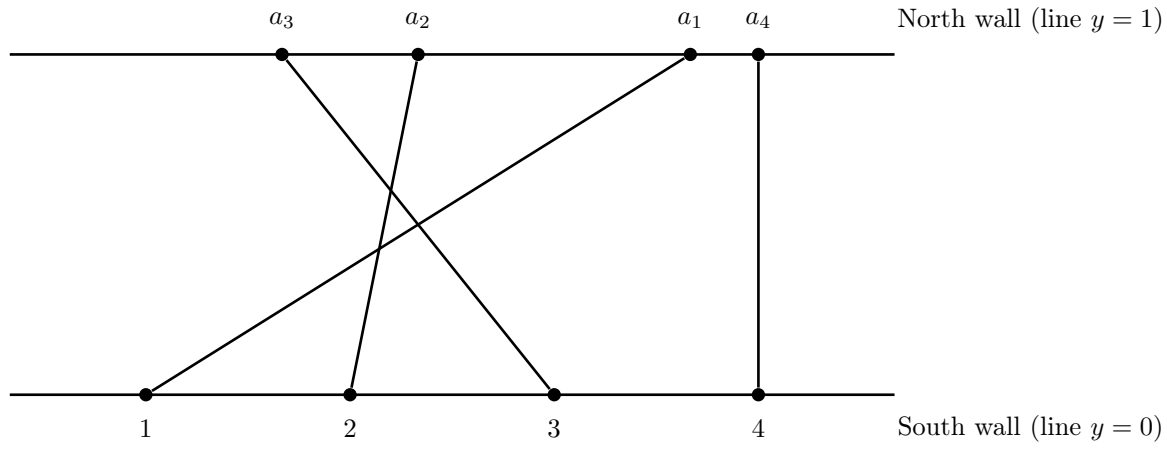


Figure 3. An example problem instance with $n = 4$. The number of intersection points is 3. Note that only the x-coordinates of the climbers are shown, as all climbers on the North wall have their y-coordinates equal to 1, and climbers on the South wall have their y-coordinates equal to 0.

For full credit your algorithm should run in $O(n \log n)$ time. *Hint: the problem can be solved using an adaptation of merge-sort.*

Solution: Notice that we only need to count the number of pairs i, j such that $i < j$ and $a_j < a_i$. We call such a pair an inversion, and thus, we need to count the number of inversions in the array $A = [a_1, \ldots, a_n]$.

A naive way of doing this would be going through all $1 \leq i < j \leq n$ and checking whether $a_i > a_j$, but this would need $\Theta(n^2)$ comparisons. However, we already know that we can sort the array A using $\Theta(n \log n)$ comparisons. If we carefully observe what happens in the sorting, (i.e., how the order of the elements get changed over time,) we can figure out how many inversions

were there to begin with. We can modify the divide and conquer sorting algorithm, MergeSort, for this purpose.

To see this, suppose that we have already sorted the first half of A and the second half of A separately, and counted the number of inversions that were in those portions separately. Formally, suppose that we have an array $C = [c_1, \ldots, c_{\frac{n}{2}}, c_{\frac{n}{2}+1}, \ldots, c_n]$ such that, $c_1 < c_2 < \cdots < c_{\frac{n}{2}}$ and $c_{\frac{n}{2}+1} < c_{\frac{n}{2}+2} < \cdots < c_n$. Now, while merging the two halves into a single sorted array, we can count the number of remaining inversions, i.e. inversions that involve one element from the left half and one element from the right half. Note that the total number of such inversions is the sum over all elements of the right half of the number of elements in the left half that are larger. We can compute this sum in linear time during the merge step. Consider the merge procedure in MergeSort (see Algorithm 3 below for the modified version) and suppose that we have merged $i-1$ elements from the first half and $j-1$ elements form the second half, and are now comparing c_i and $c_{\frac{n}{2}+j}$. We will record the contribution of every element of the right half of C to the number of inversions once it is added to the merged array. If $c_i < c_{\frac{n}{2}+j}$, there is nothing to be done, since c_i is added to the merge array, not $c_{\frac{n}{2}+j}$. If $c_{\frac{n}{2}+j} < c_i$, we add $\frac{n}{2} - i + 1$ to our counter, since $c_{\frac{n}{2}+j}$ must be put before all the remaining elements of the first half, $c_i, c_{i+1}, \ldots, c_{\frac{n}{2}}$, and thus participates in exactly $\frac{n}{2} - i + 1$ more inversions with an element of the left half of C. Hence, we have the following modified MergeSort algorithm for counting the number of inversions (see Algorithm 2). We get the answer by calling RouteIntersections$(A, 1, n)$.

Algorithm 2 Modified MergeSort

```
1: function RouteIntersections(X, p, r)
2:     answer ← 0
3:     if p < r then
4:         q ← ⌊(p+r)/2⌋
5:         answer ← answer + RouteIntersections(X, p, q)
6:         answer ← answer + RouteIntersections(X, q + 1, r)
7:         answer ← answer + RouteIntersectionsMerge(X, p, q, r)
8:     end if
       return answer
9: end function
```

Algorithm 3 Modified merging procedure.

```
1: function RouteIntersectionsMerge(X, p, q, r)
2:     answer ← 0
3:     n_1 ← q + 1 − p
4:     n_2 ← r − q
5:     Create new arrays L[1, ..., n_1 + 1], and R[1, ..., n_2 + 1]
6:     for i = 1, ..., n_1 do
7:         L[i] ← X[p + i − 1]
8:     end for
9:     L[n_1 + 1] ← ∞
10:    for j = 1, ..., n_2 do
11:        R[j] ← X[q + j]
12:    end for
13:    R[n_2 + 1] ← ∞
14:    i ← 1
15:    j ← 1
16:    for k = p, ..., r do
17:        if L[i] < R[j] then
18:            X[k] ← L[i]
19:            i ← i + 1
20:        else
21:            answer ← answer + (n_1 − i + 1)
22:            X[k] ← R[j]
23:            j ← j + 1
24:        end if
25:    end for
       return answer
26: end function
```