Introduction to Algorithms
3rd Edition

—

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, & Clifford Stein

Graham Strickland

August 7, 2025

1 The Role of Algorithms in Computing

1.1 Algorithms

1.1-1

Suppose we need to find the smallest number of buildings in a suburb of a city which need security huts for surveillance of vehicles entering the suburb. We can model this problem by trying to find the convex hull, where buildings are vertices are the n points in the plane.

1.1-2

Space required for data storage, since memory is finite in any computing system executing an algorithm.

1.1-3

Linked lists have the following strengths:

- They are easy to insert items into.
- They are easy to remove items from.
- They do not require a large block of contiguous memore to be allocated ahead of insertion.

They have the following limitations:

- Traversing the linked list is slow.

- Accessing an element is not easy and requires indirect methods.
- Deallocating the memory used is slow.

1.1-4

The traveling-salesman problem can be interpreted as a series of many different shortest-path problems which are combined into one. However, the shortest-path problem does not require that whoever traverses the path return to where they started. It can also be solved efficiently, whereas the traveling-salesman problem cannot.

1.1-5

If we are tring to find the optimal layout of train routes so that trains do not cross tracks at the same time, then only the best solution will do. On the other hand, if we are trying to minimize the cost of procuring parts for the train, then an approximation to the best solution may be good enough.

1.1 Algorithms as a technology

1.2-1

A Google search uses algorithmic content to find matches to a seqrch query in the shortest time possible. The algorithms used must find the shortest path between various networks of web pages as well as parse through large amounts of text data in the shortest time possible.

1.2-2

We are looking for the interval where

$$\begin{aligned} 8n^2 &< 64n \lg n \\ \Rightarrow n &< 8 \lg n \\ \Rightarrow \frac{n}{\lg n} &< 8 \\ \Rightarrow -n \lg n &< 8 \\ \Rightarrow n \lg n &\geq 8. \end{aligned}$$

For $n < 44$, we have insertion sort faster than merge sort.

1.2-3

We are looking for the interval where $100n^2 < 2^n$, thus for $n = 15$, n^2 runs faster than 2^n.

Problems

1-1

From the program in `rs/clrs_compruntime/src/main.rs`, we have the following output:

```
-------------------------------------------------------------
f(n)            1 s             1m              1h
-------------------------------------------------------------
lg(n)                inf             inf             inf
sqrt(n)   1.00000000e+12  3.60000000e+15  1.29600000e+19
n                1000000        60000000  3.60000000e+09
nlg(n)             62746         2801417       133378058
n^2                 1000            7745           60000
n^3                   99             391            1532
2^n                   19              25              31
n!                    10              12              13
-------------------------------------------------------------
-----------------------------------------------------------------
      1d              1m              1y              1c
-----------------------------------------------------------------
           inf             inf             inf             inf
7.46496000e+21  6.71846400e+24  9.94519296e+26  9.94519296e+30
8.64000000e+10  2.59200000e+12  3.15360000e+13  3.15360000e+15
2.75514751e+09  7.18708564e+10  7.97633893e+11  6.86109568e+13
        293938         1609968         5615692        56156922
          4420           13736           31593          146645
            36              41              44              51
            14              16              17              18
-----------------------------------------------------------------
```

In generating the above results, we have made use of the following two Rust functions in `rs/clrs_algorithms/src/big_o.rs`:

```
pub fn inverse_nlogn(x: f64) -> f64 {
    let max_iters = 10;
    let mut a_0 = x / (x.log2());
    let mut a_1: f64 = 0.0;

    for _ in 0..max_iters {
        a_1 = a_0 - (a_0 * a_0.log2() - x) / ((1.0 / LN_2) + a_0.log2());
        if ((a_1 * a_1.log2()) - (a_0 * a_0.log2())).abs() < 1.0 {
            return a_1.floor();
        } else {
            a_0 = a_1;
        }
    }

    a_1.floor()
}
```

and:

```
pub fn inverse_factorial(x: f64) -> f64 {
    let mut i = 0.0;
    let mut fact = 1.0;

    while fact < x {
        if i > 0.0 {
            fact *= i;
        }
        i += 1.0;
    }

    i - 1.0
}
```

`inverse_nlogn` makes use of Newton's method to calculate an approximate value $m(x)$ s.t.

$$n \lg n = x \Rightarrow m(x) \approx n,$$

while `inverse_factorial` generates an approximate value $m(x)$ s.t.

$$x = n! \Rightarrow m(x) = \lfloor n \rfloor.$$

2 Getting Started

2.1 Insertion sort

2.1-1

In Figure 1, we illustrate the operation of INSERTION-SORT on the array $A = \langle 31, 41, 59, 26, 41, 58 \rangle$.

2.1-2

INSERTION-SORT(A)

1: **for** $j = 2$ **to** $A.length$ **do**
2: $key = A[j]$
3: // Insert $A[j]$ to the sorted sequence $A[1 \,..\, j-1]$
4: $i = j - 1$
5: **while** $i > 0$ **and** $A[i] < key$ **do**
6: $A[i+1] = A[i]$
7: $i = i - 1$
8: $A[i+1] = key$

The implementation can be seen in the following file: `rs/clrs_algorithms/src/sorting.rs`.

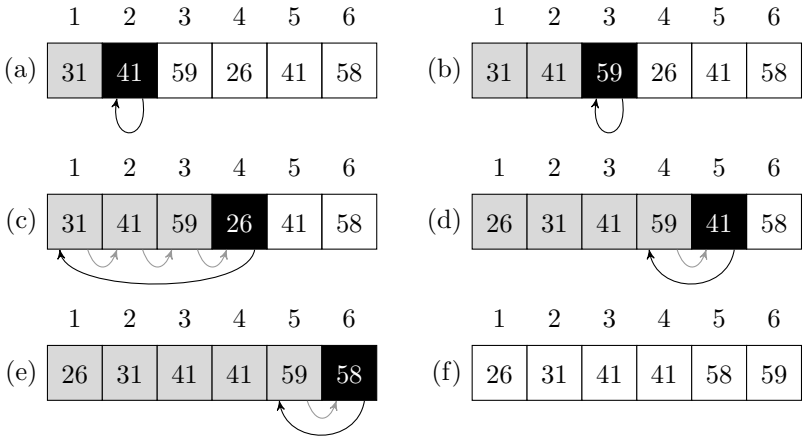

Figure 1: Illustration of INSERTION-SORT.

2.1-3

LINEAR-SEARCH(A, ν)
1: $j = 1$
2: **while** $j \neq A.length$ **do**
3: **if** $A[j] == \nu$ **then**
4: **return** j
5: **else**
6: $j = j + 1$
7: **return** NIL

Initialisation: With the loop invariant being that $A[j]$ refers to an element of the sequence $A = \langle a_1, a_2, \ldots, a_n \rangle$ and each element in $A[1 \,..\, j-1]$ has been checked for equality, we have the loop invariant valid, since $j = 1$ and $A[1]$ is in A for $n \geq 1$, at the start of the **while** loop 1 – 6.

Maintenance: Either the **if** statement in line 3 returns $A[j]$ or the **else** statement in line 5 increments j by 1, so that after each pass of the **while** loop, we have checked whether or not $A[j] = \nu$, and the condition of the **while** loop ensures that $A[j]$ is in A.

Termination: If the **while** loop terminates, then either the **if** condition in line 3 is true, so that $A[j] = \nu$ or $j = A.length$, at which point we return the special value NIL.

Thus the algorithm is correct, since the loop invariant is initialised and maintained throughout, and the algorithm terminates with the correct output; if a value is returned, ν is in A, and occurs at $A[j]$ for return value j, otherwise the special value NIL was returned, and ν is not in A.

The implementation can be seen in the following file:
`rs/clrs_algorithms/src/search.rs`.

2.1-4

We have the following addition problem:

Input: Two n-element arrays A and B containing binary digits.

Output: $(n+1)$-element array C containing the sum of A and B.

We have the following algorithm as a solution:

```
BINARY-ADDITION(A, B, n)
 1: carry = 0
 2: for i = 1 to n do
 3:    if A[i] = 1 and B[i] = 1 then
 4:       if carry == 0 then
 5:          C[i] = 0
 6:          carry = 1
 7:       else
 8:          C[i] = 1
 9:          carry = 1
10:    else if A[i] == 1 and B[i] == 0 or
             A[i] == 0 and B[i] == 1 then
11:       if carry == 0 then
12:          C[i] = 1
13:       else
14:          C[i] = 0
15:          carry = 1
16:    else
17:       C[i] = carry
18:       carry = 0
19: C[n + 1] = carry
20: return C
```

The implementation can be seen in the following file:
`rs/clrs_algorithms/src/binary.rs`.

2.2 Analyzing algorithms

2.2-1

$$\frac{n^3}{1000} - 1000n^2 - 100n + 3 \approx \Theta(n^3).$$

2.2-2

```
SELECTION-SORT(A)
1: for i = 1 to A.length − 1 do
```

2: $smallest = i$
3: **for** $j = i$ **to** $A.length$ **do**
4: **if** $A[j] < A[smallest]$ **then**
5: $smallest = j$
6: $A[smallest] = A[i]$

The **for** loop from lines 2-8 maintains the loop invariant that $A[1 \,..\, i-1]$ contains sorted elements in ascending order.

The algorithm only needs to run for the the first $n-1$ elements since the nth element will already be the largest element in the array after each smaller element in $A[1 \,..\, n]$ has been sorted.

Since there is no **while** loop and each search for the smallest item in the subarray $A[i+1 \,..\, n-1]$ must occur, the best and wors-case running times are the same, i.e., $\Theta(n^2)$.

The implementation can be seen in the following file:
`rs/clrs_algorithms/source/sorting.rs`

2.2-3

Assuming the element being searched for is equally likely to be at any position in the array, we would on average search through

$$\frac{1+2+\cdots+n}{n} = \frac{1}{n}\sum_{k=1}^{n} k = \frac{n+1}{2}$$

elements for an input sequence of length n. In the worse case we would search through all n elements.

Thus the average- and worst-case running times are $\Theta(n)$, since

$$\frac{n+1}{2} = \frac{n}{2} + \frac{1}{2},$$

and the dominant term is $\frac{1}{2}n$, which is equivalent to $\Theta(n)$. Obviously, looping through n items sequentially is $\Theta(n)$, so the average- and worse-case running times are in an equivalent Θ-class.

2.2-4

By only inputting elements that are equivalent or close to the expected output.

2.3 Designing algorithms

2.3-1

In Figure 2, we illustrate the operation of MERGE-SORT on the array $A = \langle 3, 41, 52, 26, 38, 57, 9, 49 \rangle$.

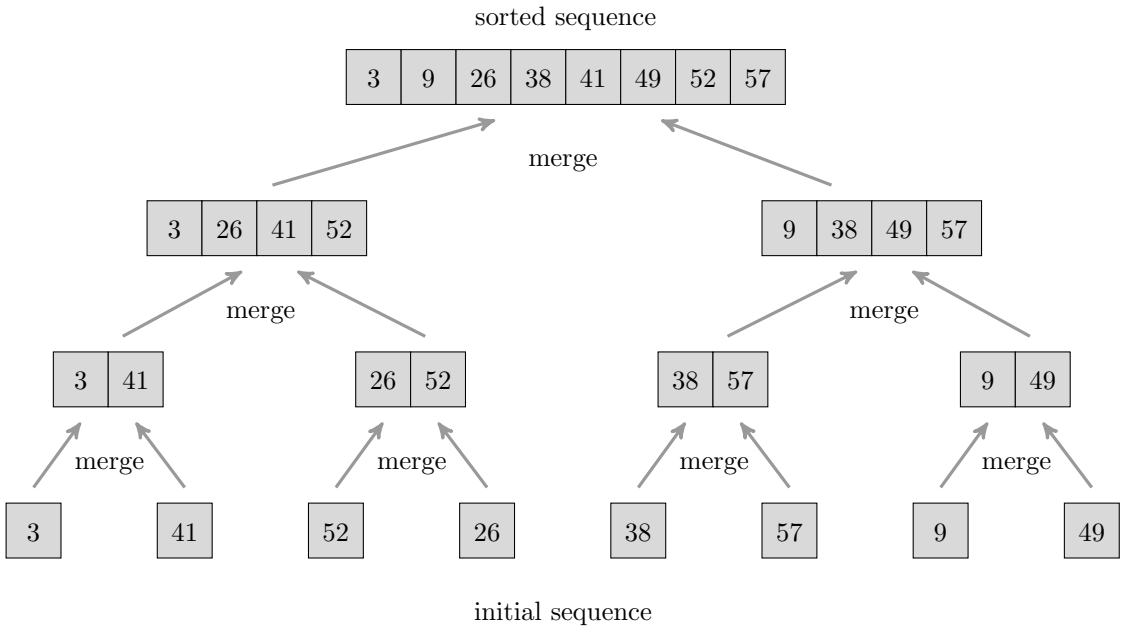

Figure 2: Illustration of MERGE-SORT.

2.3-2

MERGE(A, p, q, r)
1: $n_1 = q - p + 1$
2: $n_2 = r - q$
3: let $L[1..n_1 + 1]$ and $R[1..n_2 + 1]$ be new arrays
4: **for** $i = 1$ **to** n_1 **do**
5: $\quad L[i] = A[p + i - 1]$
6: **for** $j = 1$ **to** n_2 **do**
7: $\quad R[j] = A[q + j]$
8: $i = 1$
9: $j = 1$
10: **for** $k = p$ **to** r **do**
11: $\quad$ **if** $i \leq n_1 + 1$ and $L[i] \leq R[j]$ **then**
12: $\quad\quad A[k] = L[i]$
13: $\quad\quad i = i + 1$
14: $\quad$ **else if** $A[k] == R[j]$ **then**
15: $\quad\quad j = j + 1$

The implementation can be seen in the following file:
`rs/clrs_algorithms/src/sorting.rs`

2.3-3

Proof. We have the base case for $n = 2^1$ given by

$$T(n) = T(2) = 2 = 2\log_2 2^1 = 2 \cdot 1.$$

Now, we assume as our induction step that if $n = 2^k$ for $k > 1$,

$$T(n) = 2T(n/2) + n = n \lg n.$$

Then, suppose that $n = 2^{k+1}$, so that we have

$$\begin{aligned}
T(n) &= 2T(2^{k+1}/2) + 2^{k+1} \\
&= 2T(2^k) + 2^{k+1} \\
&= 2(2^k \lg 2^k) + 2^{k+1} && \text{(by our induction assumption)} \\
&= 2^{k+1} \lg 2^k + 2^{k+1} \lg 2 && \text{(since } \log_2 2 = 1\text{)} \\
&= 2^{k+1} \lg 2^{k+1} && \text{(since } \log_2 a + \log_2 b = \log_2 ab\text{)}
\end{aligned}$$

Therefore, by induction on $n = 2^k$, $T(n) = n \lg n$. □

2.3-4

We let $T(n)$ be the running time of a problem of size n. For $n \leq 2$, we have $T(n) = \Theta(2) = \Theta(1)$, since this is just the process of inserting an item in either position 1 or 2, so a worst-case time is $\Theta(1)$.

If $n > 2$, our division of the problem yields $n-1$ subproblems, each of which is $\frac{n-1}{n}$ times the size of the original. It takes $T(\frac{n(n-1)}{n}) = T(n-1)$ to solve one subproblem, so it takes $(n-1)T(n-1)$ to solve $n-1$ of them. It then takes $\Theta(n)$ time to divide the problem into subproblems and insert the last item in the array into $A[1\,..\,n-1]$, so we have

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq 2 \\ (n-1)T(n-1) + \Theta(1) & \text{otherwise} \end{cases}.$$

2.3-5

BINARY-SEARCH(A, ν)

```
1: low = 1
2: high = A.length
3: while low < high do
4:     mid = ⌊(high+low)/2⌋
5:     if A[mid] == ν then
6:         return mid
7:     else if A[mid] > ν then
8:         high = mid − 1
9:     else
```

10: $low = mid + 1$
11: **return** -1

Suppose we have an array A of length n. Then, in the worst case, we loop through the **while** loop of BINARY-SEARCH (lines 3-9) until $low = high$ and return -1, so that ν is not in A. If we continually divide n by 2, and n is a power of 2, then we will execute

$$\lfloor \frac{high - low}{2} \rfloor$$

exactly $\lg n$ times. If n is not a power of 2, then the same operation executes $\lfloor \lg n \rfloor$ times. In any case, we have a worst-case running time of $\Theta(\lg n)$ for BINARY-SEARCH.

The implementation can be seen in the following file:
`rs/clrs_algorithms/src/search.rs`.

2.3-6

Since the loop invariant of INSERTION-SORT guarantees that we have $A[1..j-1]$ sorted, we can use the BINARY-SEARCH algorithm of 2.3-5 in place of the **while** loop of lines 5-7.

Since every other operation in the algorithm for INSERTION-SORT is $\Theta(n)$ and the **while** loop is nested but also $\Theta(n)$, we can improve the overall worst-case running time to $\Theta(n \lg n)$, which is better than $\Theta(n^2)$ for the original INSERTION-SORT algorithm.

2.3-7

Suppose we take the recursive approach of dividing the set S of n integers in half and checking each half. If the first half has index 1 and index $\lfloor n/2 \rfloor$ referencing elements such that for $A = S$ enumerated as an array, $A[1] = A[\lfloor n/2 \rfloor] = x$, we return *true*. If not, we check the other half with the same condition and if the condition is *false* for the upper half, wle recursively check each lower and upper half of the two subarrays. This process of repeated recursive subdivisions will have a running time of $\Theta(n \lg n)$, since the recurrence is the same as for merge sort.

Problems

2-1

a. *Proof.* Since INSERTION-SORT has a running time of $\Theta(n^2)$, due to the nested **while** loop of lines 5-7 (p. 26), and the loop of lines 1-7, if we are instead sorting n items divided into n/k sublists, the outer **for** loop will run k times for each sublist. Since each sublist will contain k items, the **while** loop will run $k - 1$ times for each sublist, so we have a running time of $\Theta(k^2)$ for each sublist, and since there are n/k sublists, the total

running time will be

$$\Theta\left(\frac{n}{k} \cdot k^2\right) = \Theta(nk).$$

□

b. If we merge the sublists when their length reaches k, we are removing $\lg k$ levels from the recursion tree on p. 38, so we have

$$\lg n - \lg k = \lg n/k$$

levels on the tree. Now, from **a.**, we have the INSERTION-SORT algorithm on the n/k sublists with a running time of $\Theta(nk)$, so the new algorithm will have a running time of

$$\Theta(n \lg (n/k) + nk) = \Theta(n[\lg (n/k) + k]) = \Theta(n \lg (n/k)).$$

c. We are looking for the point at which

$$\begin{aligned} \Theta(nk + n \lg (n/k)) &= \Theta(n \lg n) \\ \Leftrightarrow c_1[nk + n \lg (n/k)] &= c_2 n \lg n \\ \Leftrightarrow c_2 \lg n &= c_1[k - \lg n/k] \\ \Leftrightarrow k + \lg k &= \frac{(c_1 + c_2) \lg n}{c_2}. \end{aligned}$$

d. In practice, we should choose k to be $\lceil \lg n \rceil$, since this will be the closest point to which

$$k + \lg k = \frac{(c_1 + c_2) \lg n}{c_2},$$

where $k \in \mathbb{Z}$.

2-2

a. We need to prove a loop invariant for each loop in the algorithm.

b. *Proof.* At the start of each iteration of the **for** loop of lines 2-4, $A'[1 \,..\, i]$ consists of elements sorted in ascending order.

Initialisation: For the first loop of lines 2-4, $A'[1 \,..\, i]$ is a one-element array, $A'[1]$, which is (trivially) sorted.

Maintenance: After each **for** loop, the first out-of-order element from $A.length$ down to $i + 1$ has been swapped with each element before it until it is in the correct position in $A'[1 \,..\, i]$.

Termination: The **for** loop terminates when $j == i + 1$, so that, at that point, $A'[i] < A'[i + 1]$ and the loop invariant holds for $A'[1 \,..\, i + 1]$.

□

c. *Proof.* For each iteration of the **for** loop of lines 1-4, $A'[1\,..\,i+1]$ is sorted in ascending order.

Initialisation: At initialisation, $i = 1$, so $A'[1\,..\,i+1]$ is $A'[1\,..\,2]$ and the termination of **b.** guarantees that this will be sorted.

Maintenance: Each execution of the **for** loop of lines 2-4 sorts $A'[1\,..\,i+1]$, so the loop invariant is maintained.

Termination: The loop terminates when $i == A.length - 1$, so $A'[1\,..\,i+1] = A'[1\,..\,n]$ has been sorted.

□

d. The worst-case running time occurs when A is sorted in descending order, so the **for** loop of lines 1-4 executes $n-1$ times and the **for** loop of lines 2-4 executes $n-i+1$ times. Thus we have $(n-1)(n-i+1) = \Theta(n^2)$ worst-case running time, equivalent to that of INSERTION-SORT.

The implementation can be seen in the following file:
`rs/clrs_algorithms/src/sorting.rs`.

2-3

a. The running time is $\Theta(n)$, since there is only one **for** loop that runs n times.

b. HORNERS-RULE$(a_1\,..\,a_n, n, x)$

```
1: y = 0
2: for i = n downto 0 do
3:     temp = x
4:     for j = i downto 0 do
5:         temp = temp · x
6:     y = a_i · temp + y
7: return y
```

The running time of this algorithm is $\Theta(n^2)$, since the outer **for** loop of lines 2-6 runs n times and for each execution of this loop, the inner **for** loop of lines 4-5 runs $n-i$ times, where i is the value of the loop iteration variable declared in line 2 of that iteration.

Cleary this algorithm has a much greater running time than Horner's rule.

c. *Proof.* **Initialisation:** The initialisation of the **for** loop of lines 2-3 assigns i to n, thus

$$\sum_{k=0}^{n-(i+1)} a_{k+i+1}x^k = \sum_{k=0}^{n-(n+1)} a_{k+n+1}x^k = \sum_{k=0}^{-1} a_{k+n+1}x^k = 0,$$

which is valid, since $y = 0$ upon initialisation of the **for** loop, therefore its initialisation is correct.

Maintenance: For each subsequent iteration of the **for** loop, y is assigned to $a_i + x \cdot y$, so that, for the first **for** loop, y is assigned to $a_1 + x \cdot 0 = a_1$, and

$$\sum_{k=0}^{n-(n-1+1)} a_{k+(n-1)+1} x^k = \sum_{k=0}^{0} a_{k+n} x^k = a_1,$$

which is correct.

For subsequent **for** loops, $a_i + x \cdot y$ multiplies the value of the previous **for** loop,

$$\sum_{k=0}^{n-(i+2)} a_{k+i+2} x^k$$

by x, so that we have

$$x \sum_{k=0}^{n-i-2} a_{k+i+2} x^k = \sum_{k=0}^{n-i-2} a_{k+i+2} x^{k+1},$$

and we add a_i to this value, with the result

$$y = \sum_{k=0}^{n-(i+1)} a_{k+i} x^{k+1}.$$

Thus the loop invariant is maintained at each iteration.

Termination: Since the **for** loop terminates at $i = 0$, we have

$$\sum_{k=0}^{n-(i+1)} a_{k+i+1} x^{k-1} = \sum_{k=0}^{n-1} a_{k+1} x^{k-1}$$

at the beginning of the lop, and with one more operation,

$$y = \sum_{k=0}^{n} a_k x^k,$$

so that the loop terminates correctly.

□

d. Since the loop invariant is initialised, maintained, and terminates correctly, for

$$P(x) = \sum_{k=0}^{n} a_k x^k,$$

the correct evaluation is performed for $a_0, a_1, \ldots, a_n$.

The implementation of Horner's rule can be seen in the following file: `rs/clrs_polynomials/src/lib.rs`.

2-4

a. $(1,5),(2,5),(3,5),(4,5)$, and $(3,4)$.

b. The array containing all elements of the set $\{1,2,\ldots,n\}$ in reverse order has the most possible inversions. Since there are $n-1$ inversions, starting with 1, $n-2$ starting with 2, ..., there are

$$\sum_{k=1}^{n-1} k$$

inversions, or

$$\frac{n(n-1)}{2}$$

inversions.

c. Since the worst-case running time of INSERTION-SORT occurs for exactly the array discussed in b. above, and we know that INSERTION-SORT has a worst-case running time which is $\Theta(n^2)$, we see that there is a direct correspondence between the number of inversions in the input array of INSERTION-SORT and its running time. This follows logically from the fact that each element from $j = 2$ to $A.length$ (where A is the input array) must be sorted if there exists an inversion with j as its second coordinate.

d. As with MERGE-SORT, we have two subalgorithms, one which finds the number of inversions in a given subarray as follows:

MERGE-INVERSIONS(A, p, q, r)

```
1:  n_1 = q - p + 1
2:  n_2 = r - q
3:  let L[1..n_1 + 1] and R[1..n_2 + 1] be new arrays
4:  for i = 1 to n_1 do
5:      L[i] = A[p + i - 1]
6:  for j = 1 to n_2 do
7:      R[j] = A[q + j]
8:  L[n_1 + 1] = ∞
9:  R[n_2 + 1] = ∞
10: i = 1
11: j = 1
12: inversions = 0
13: counted = FALSE
14: for k = p to r do
15:     if counted == FALSE and R[j] < L[i] then
16:         inversions = inversions + n_1 - i + 1
17:         counted = TRUE
18:     if L[i] ≤ R[j] then
19:         A[k] = L[i]
```

```
20:       i = i + 1
21:    else
22:       j = j + 1
23:       counted = FALSE
24: return inversions
```

Now that we have the recursive procedure that separates the array into subarrays and finds the number of inversions in each, we find the total number of inversions using the following algorithms:

```
COUNT-INVERSIONS(A, p, r, inversions)
1: if p < r then
2:    q = ⌊(p + r)/2⌋
3:    inversions = COUNT-INVERSIONS(A, p, q, inversions) + inversions
4:    inversions = COUNT-INVERSIONS(A, q+1, r, inversions)+inversions

5:    inversions = MERGE-INVERSIONS(A, p, q, r) + inverions
6: return inversions
```

Since the algorithm is based entirely upon MERGE-SORT, with almost the same number of steps, it also has $\Theta(n \lg n)$ running time.

The implementation of COUNT-INVERSIONS can be seen in the following file: `rs/clrs_algorithms/src/arrays.rs`.