CHAPTER 2

Sorting Algorithms

II.1. Introduction

One of the fundamental problems in computer science is sorting a list of items. There are a large number of solutions to this issue, known as sorting algorithms. In this chapter, we try to describe the different sorting algorithms that have been considered the most widely used in different applications. Sorting is the process of arranging or rearranging the elements of a list in ascending or descending order, which can be numerical, lexicographical or any other order defined by the user. The output is a permutation or reordering of the input.

II.2. Complexity of Sorting Algorithms

The complexity of the sorting algorithm measures the execution time as a function of the number $\boldsymbol{n}$ of items to be stored. Each sorting algorithm $\boldsymbol{S}$ will be composed of the following operations, where $\boldsymbol{A_1, A_2, A_3 \cdots, A_n}$ contains the items to be sorted and $\boldsymbol{B}$ is an auxiliary location.

- Comparisons, which test whether $\boldsymbol{A_i < A_j}$ or test whether $\boldsymbol{A_i < B}$.
- Exchanges that swap the contents of $\boldsymbol{A_i}$ and $\boldsymbol{A_j}$ or $\boldsymbol{A_i}$ and $\boldsymbol{B}$.
- Assignment that defines $\boldsymbol{B}$: $\boldsymbol{A_i}$ then defines $\boldsymbol{A_j \leftarrow B}$ or $\boldsymbol{A_j \leftarrow A_i}$.

Normally, the complexity function measures only the number of comparisons, since the number of other operations is at most a constant factor of the number of comparisons.

II.3. Types of Sorting Algorithms

Sorting algorithms can be classified in two types :

1) **Internal Sorting:** This method uses only the primary memory during the sorting process. All data items are held in main memory and no secondary memory is required for this sorting process. Internal Sorting is limited; they can only process a small amount of data due to memory constraints. The time required for reading or writing is not considered significant when assessing the performance of internal sorting methods. E.g., Bubble Sort, Insertion Sort, Selection Sort, Shell sort, Quick Sort, Radix Sort, Heap Sort, Two-way Merge Sort etc.
2) **External sorting:** In contrast to internal sorting, external sorting is used when a large quantity of data needs to be sorted. This process uses external memory such as HDD to store data that cannot be stored in main memory. So, the main memory only contains the data currently being sorted. All external sorts are based on the merge process. Read and write access times are a major concern in determining the sorting performance of these methods. E.g., Merge Sort, Multiway Merge, Polyphase merge.

II.4. Bubble Sort

The bubble sort is the oldest and simplest sort in use. Unfortunately, it's also the slowest. The bubble sort works by comparing each item in the list with the item next to it, and swapping them if required. The algorithm repeats this process until it has gone through the entire list without swapping any items (in other words, all items are in the correct order). A C++ implementation of the standard bubble sort algorithm for a vector of integers is as follows:

```
void bubble_sort (int Vec[], int n) { // Bubble Sort
      for (int i=0; i<n-1; i++)  // Bubble up ith record
            for (int j=0; j<n-i-1; j++)
                  if (Vec[j]> Vec[j+1])
                        swap(Vec[j], Vec[j+1]);
}
```

Example:

Let's look at how bubble sort works using the following unsorted array: $[\mathbf{5}, \mathbf{3}, \mathbf{9}, \mathbf{1}, \mathbf{7}, \mathbf{8}, \mathbf{2}, \mathbf{4}, \mathbf{6}]$. The bubble sort algorithm looks at two adjacent elements at a time, swapping them if they are out of order. At the end of each pass, the largest element "bubbles" to the end of the vector, as shown below:

First pass:

[5, 3, 9, 1, 7, 8, 2, 4, 6]	5 and 3 are out of order, so swap!
[3, 5, 9, 1, 7, 8, 2, 4, 6]	5 and 9 are in the correct order, so no swap needed.
[3, 5, 9, 1, 7, 8, 2, 4, 6]	9 and 1 are out of order, so swap!
[3, 5, 1, 9, 7, 8, 2, 4, 6]	9 and 7 are out of order, so swap!
[3, 5, 1, 7, 9, 8, 2, 4, 6]	9 and 8 are out of order, so swap!
[3, 5, 1, 7, 8, 9, 2, 4, 6]	9 and 2 are out of order, so swap!
[3, 5, 1, 7, 8, 2, 9, 4, 6]	9 and 4 are out of order, so swap!
[3, 5, 1, 7, 8, 2, 4, 9, 6]	9 and 6 are out of order, so swap!
[3, 5, 1, 7, 8, 2, 4, 6, **9**]	First pass complete! 9 is fixed in position.

Because **9** is the largest element in the array, it ends up bubbling to the end. After the first pass, we will keep **9** fixed (since we know that **9** is now in the right position) and complete another pass of bubble sort. During this second iteration, we will bubble the second largest element up to its correct position.

Second pass:

[3, 5, 1, 7, 8, 2, 4, 6, **9**]	3 and 5 are in the correct order, so no swap needed.
[3, 5, 1, 7, 8, 2, 4, 6, **9**]	5 and 1 are out of order, so swap!
[3, 1, 5, 7, 8, 2, 4, 6, **9**]	5 and 7 are in the correct order, so no swap needed.
[3, 1, 5, 7, 8, 2, 4, 6, **9**]	7 and 8 are in the correct order, so no swap needed.
[3, 1, 5, 7, 8, 2, 4, 6, **9**]	8 and 2 are out of order, so swap!
[3, 1, 5, 7, 2, 8, 4, 6, **9**]	8 and 4 are out of order, so swap!
[3, 1, 5, 7, 2, 4, 8, 6, **9**]	8 and 6 are out of order, so swap!
[3, 1, 5, 7, 2, 4, 6, **8**, **9**]	Second pass complete! 8 is fixed in position.

We can repeat this process by conducting multiple passes until the array is fully sorted. What is the time complexity of bubble sort?

Time Complexity of Bubble Sort

If we define the size of the array as $\boldsymbol{n}$ then we complete $\boldsymbol{n-1}$ comparisons on the first pass, $\boldsymbol{n-2}$ comparisons on the second pass, $\boldsymbol{n-3}$ comparisons on the third pass, and so on.

The total number of comparisons we complete is equal :

$$\boldsymbol{T(n) = (n-1) + (n-2) + \cdots + 1 = \frac{n(n-1)}{2} = \mathcal{O}(n^2)}$$

Thus, our current bubble sort implementation runs in $\boldsymbol{\mathcal{O}(n^2)}$ time.

Remark: This specific approach is not the only way to complete a bubble sort: you can also iterate from the back to the front, instead of from the front to the back. In this alternative method, the smallest values get bubbled to the front of the array.

II.5. Selection Sort

Selection sorting is one of the simplest sorting algorithms: First, find the smallest item in the array and swap it with the first entry (itself if the first entry is already the smallest). Then find the next smallest item and swap it with the second entry. Continue in this way until the entire array has been sorted. The code is shown below:

```
void selection_sort(int Vec[], int n) {
      for (int i = 0; i < n - 1; i++) {
            int min_index = i;
            for (int j = i + 1; j < n; j++) {
                  if (Vec[j] < Vec[min_index]) {
                        min_index = j;
                  }
            }
            swap(Vec[i], Vec[min_index]);
      }
}
```

Example:

Let's look at selection sort in action, using the same array we used earlier: $[\mathbf{5,3,9,1,7,8,2,4,6}]$. With each pass, we find the next smallest value and place it in its correct position. In the process below, the two underlined elements are swapped, and bolded elements are fixed in their correct sorted position.

$[\underline{5}, 3, 9, \underline{1}, 7, 8, 2, 4, 6]$	1 is the smallest element, so swap it to index 0.
$[\mathbf{1}, \underline{3}, 9, 5, 7, 8, \underline{2}, 4, 6]$	2 is the next smallest element, so swap it to index 1.
$[\mathbf{1,2}, \underline{9}, 5, 7, 8, \underline{3}, 4, 6]$	3 is the next smallest element, so swap it to index 2.
$[\mathbf{1,2,3}, \underline{5}, 7, 8, 9, \underline{4}, 6]$	4 is the next smallest element, so swap it to index 3.
$[\mathbf{1,2,3,4}, \underline{7}, 8, 9, \underline{5}, 6]$	5 is the next smallest element, so swap it to index 4.
$[\mathbf{1,2,3,4,5}, \underline{8}, 9, 7, \underline{6}]$	6 is the next smallest element, so swap it to index 5.
$[\mathbf{1,2,3,4,5,6}, \underline{9}, \underline{7}, 8]$	7 is the next smallest element, so swap it to index 6.
$[\mathbf{1,2,3,4,5,6}, 7, \underline{9}, \underline{8}]$	8 is the next smallest element, so swap it to index 7.
$[\mathbf{1,2,3,4,5,6,7,8}, 9]$	9 is the last element, so we are done!
$[\mathbf{1,2,3,4,5,6,7,8,9}]$	

Time Complexity of Selection Sort

Similar to bubble sort, we need to make $\boldsymbol{n-1}$ comparisons on the first pass, $\boldsymbol{n-2}$ comparisons on the second pass, and so on. Thus, the total number of comparisons can be found by:

$$\boldsymbol{T(n) = (n-1) + (n-2) + \cdots + 1 = \frac{n(n-1)}{2} = \mathcal{O}(n^2)}$$

Therefore, our selection sort implementation runs in $\boldsymbol{\mathcal{O}(n^2)}$ time.

II.6. Insertion Sort

This is the sorting technique most commonly used by card players. It consists of examining the cards one by one and inserting them in their place among those already examined (thus sorting them). Insertion sorting is a simple sorting algorithm that constructs the final sorted array (or list) one item at a time. The list is divided into two parts, one composed of sorted items and the other of unsorted items. Each item of the unsorted list is considered one by one and inserted into the sorted list at its appropriate position. In each pass, the sorted list is traversed in the backward direction to find the position where the unsorted item can be inserted. The implementation of insertion sorting is illustrated below:

```
void insertion_sort(int Vec[], int n) {
      for (int i = 1; i < n; i++) {
            int current = Vec[i];
            int j = i - 1;
            while (j >= 0 && Vec[j] > current) {
                  Vec[j + 1] = Vec[j];
                  j--;
            }
            Vec[j + 1] = current;
      }
}
```

As we iterate through the array, we look at each element and compare it with the elements that come before it. If the element before the current element is out of position compared to the current element (i.e., there is an inversion), we shift that element one to the right. This allows us to identify where the current element should be placed relative to the elements that come before it.

Example:

Let's analyze insertion sort using the same unsorted array we used earlier: $[\mathbf{5,3,9,1,7,8,2,4,6}]$. We will start from the left end of the array and consider each element one at a time.

$[\underline{5}, 3, 9, 1, 7, 8, 2, 4, 6]$ 5 is in the correct position relative to the elements before it (trivially).
$[\mathbf{5}, \underline{3}, 9, 1, 7, 8, 2, 4, 6]$ Looking only at the elements before 3, where should 3 go? Before 5. At this step, we shift 5 one index to the right and insert 3 into the correct relative position, before 5.

$[\mathbf{3}, \mathbf{5}, \underline{9}, 1, 7, 8, 2, 4, 6]$ Looking only at the elements before 9, where should 9 go?

9 is in the correct position relative to 3 and 5, so no shifting is needed.

[3, 5, 9, 1**,** 7, 8, 2, 4, 6] Looking only at the elements before 1, where should 1 go? Before 3. Compared to the elements that have been visited before it, 1 should go before 3. Thus, we would shift 9 one position to the right (to index 3), 5 one position to the right (to index 2), 3 one position to the right (to index 1), and insert 1 in its correct relative position at index 0.

[1, 3, 5, 9, 7**,** 8, 2, 4, 6] Looking only at the elements before 7, where should 7 go? Before 9. Thus, we would shift 9 one position to the right (to index 4) and insert 7 in its correct relative position at index 3.

[1, 3, 5, 7, 9, 8**,** 2, 4, 6] Looking only at the elements before 8, where should 8 go? Before 9. Thus, we would shift 9 one position to the right (to index 5) and insert 8 in its correct relative position at index 4.

[1, 3, 5, 7, 8, 9, 2**,** 4, 6] Looking only at the elements before 2, where should 2 go? Before 3. Thus, we would shift 9 one position to the right (to index 6), 8 one position to the right (to index 5), 7 one position to the right (to index 4), 5 one position to the right (to index 3), 3 one position to the right (to index 2), and insert 2 in its correct relative position at index 1.

[1, 2, 3, 5, 7, 8, 9, 4**,** 6] Looking only at the elements before 4, where should 4 go? Before 5. Thus, we would shift 9 one position to the right (to index 7), 8 one position to the right (to index 6), 7 one position to the right (to index 5), 5 one position to the right (to index 4), and insert 4 in its correct relative position at index 3.

[1, 2, 3, 4, 5, 7, 8, 9, 6] Looking only at the elements before 6, where should 6 go? Before 7. Thus, we would shift 9 one position to the right (to index 8), 8 one position to the right (to index 7), 7 one position to the right (to index 6), and insert 6 in its correct relative position at index 5.

[1, 2, 3, 4, 5, 6, 7, 8, 9] The array is now fully sorted.

Time Complexity of Insertion Sort

If we define the size of the array as $\boldsymbol{n}$ then in the first pass, it will make **1** comparisons; in the second pass, it will do **2**; in the third pass, it will do **3** and so on.

Thus, the total number of comparisons can be found by:

$$\boldsymbol{T(n) = 1 + 2 + \cdots + (n-2) + (n-1) = \frac{n(n-1)}{2} = O(n^2)}$$

Thus, our current insertion sort implementation runs in $\boldsymbol{O(n^2)}$ time.

II.7. Quicksort

Quicksort is a partition-based sorting technique developed by British computer scientist Tony Hoare in 1959 and published in 1961. It is also known as partition exchange sorting. The basic concept of the quicksort process is to pick an element from an array and rearrange the other elements around it. The quicksort algorithm consists of the following two steps, which are repeated until the array is fully sorted:

1) A pivot element is selected (the method of selecting the pivot depends on the implementation, but a common strategy is to select the last element as the pivot — this is the method that will be used in these notes).

2) The remaining elements in the array are partitioned so that all elements to the left of the pivot are lesser than the pivot, and all elements to the right of the pivot are greater than the pivot.

This process is repeated recursively until sub-lists consist of only one element.

An outline of the quicksort algorithm is shown below:

```
void quick_sort(int Vec[], int left, int right) {
      if (left + 1 >= right) {
      return;
      } // if
      int pivot = partition(Vec, left, right);
      quick_sort(Vec, left, pivot);
      quick_sort(Vec, pivot + 1, right);
} // quick_sort()
int partition(int Vec[], int left, int right) {
      int pivot = right--;
      while (true) {
          while (Vec[left] < Vec[pivot]) {
               left++;
          }
          while (left < right && Vec[right - 1] >= Vec[pivot]){
               right--;
          }
          if (left >= right) {
               break;
          }
          swap(Vec[left], Vec[right - 1]);
      }
      swap(Vec[left], Vec[pivot]);
      return left;
} // partition()
```

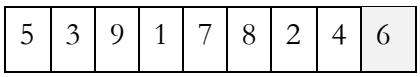
The ***partition()*** function takes in an index range [left, right) and selects the last element as the pivot. The remaining elements are partitioned so that elements to the left of the pivot are lesser, and elements to the right are greater.

Example:

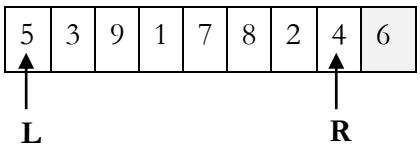
Let's look at what this code does using an example. Consider the following unsorted array:

5	3	9	1	7	8	2	4	6

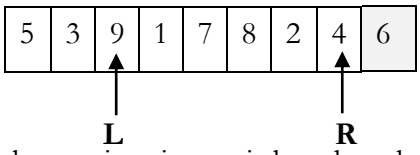
First, we will choose an arbitrary pivot in this array. In our examples, we will select the last element as our pivot:



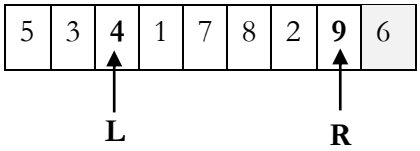
Now, we will need to partition the array so that all elements less than 6 end up to the left of 6, and all elements greater than 6 end up to the right of 6. To do so, we will keep track of two indices, left and right. The left index starts at the first element (5), while the right index starts at the last non-pivot element (4). We will refer to these two indices as L and R in the figures below.



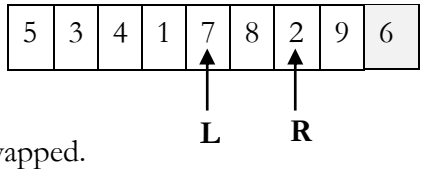
First, we increment L until the element it points to is greater than the pivot element, 6. In this case, L is incremented until it points to 9.



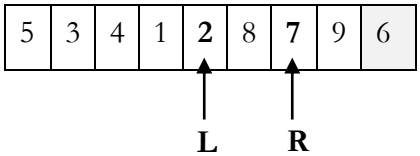
Next, we decrement R until the element it points to is less than the pivot, 6. Since the element that R is currently pointing to is already less than 6, we do not need to decrement R. Once L and R are pointing to elements that are out of place relative to the pivot, the two elements are swapped.



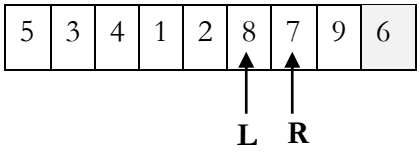
We repeat the above process until L and R meet. Increment L until it encounters a value larger than the pivot, and decrement R until it encounters a value smaller than the pivot. This happens when L is at 7 and R is at 2.



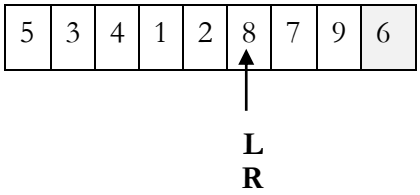
These two elements are then swapped.



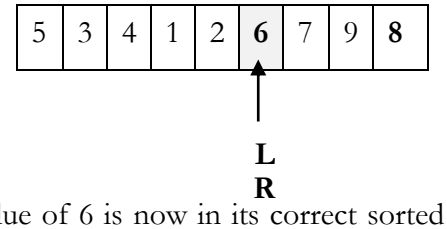
Increment L until it encounters an element greater than the pivot, 6. This happens when L is at 8:



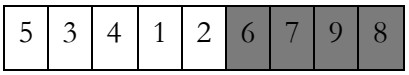
Decrement R until it encounters an element less than the pivot, 6. However, notice that the first decrement causes R to point to L.



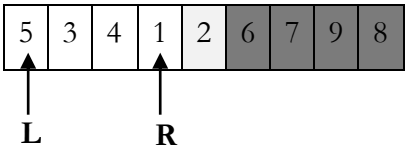
When L and R meet, swap the element at this position (in this case, 8) with the pivot value, 6.



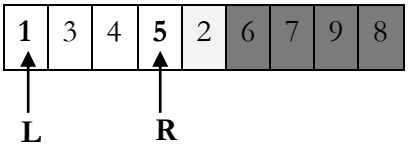
This ensures that the pivot value of 6 is now in its correct sorted position. Furthermore, all the elements to the left of 6 are smaller than 6, and all the elements to the right of 6 are larger. Since our initial array has been partitioned into two subarrays, one with elements less than 6, and one with elements greater than 6, we can recursively call quicksort on both the left and right subarrays to sort the entire array. To continue our example, let's make a recursive call and quicksort the elements to the left of 6. Note that this recursive call must run to completion before we can quicksort elements to the right of 6.



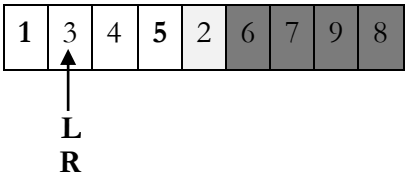
Like before, we will select the last element, 2, as our pivot, and initialize L and R:



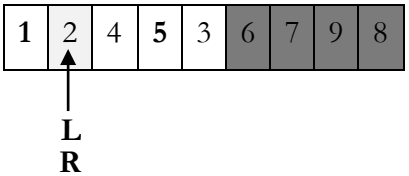
Increment L until it points to a value greater than 2, and decrement R until it points to a value less than 2 (in this case, neither L nor R needs to be moved). Then, swap the two elements.



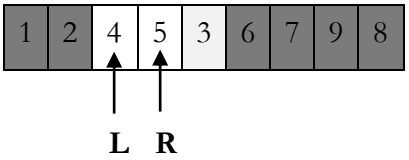
Increment L until it points to a value greater than 2, and decrement R until it points to a value less than 2. Both L and R end up at 3.



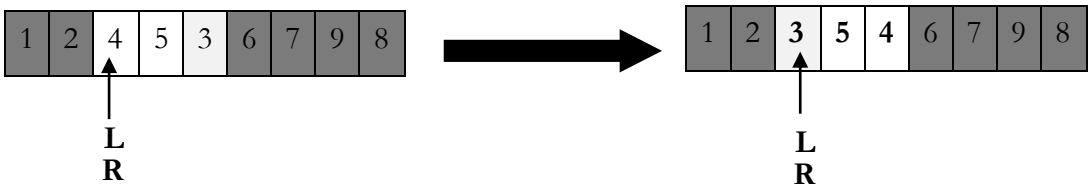
Since L and R are equal, we swap 3 with the pivot. The pivot element of 2 is now in its correct sorted position.



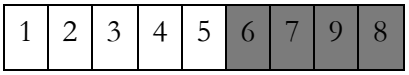
Once again, we will make recursive calls and quicksort elements to the left and right of 2. Since there is only one element to the left of 2, it must trivially be in the correct sorted position (we immediately exit in the base case). To recursively quicksort the elements to the right of 2, we select the last element (3) as the pivot.



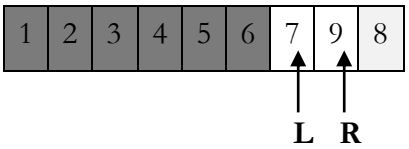
If we repeat the process of incrementing L until we reach an element greater than 3 and decrementing R until we reach an element less than 3, both L and R end up pointing to 4. We would then swap 4 with the pivot, 3:



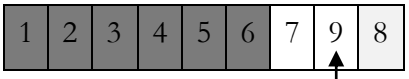
We would then recursively quicksort elements to the left and right of our pivot, 3. There are no elements to the left of 3 (since 3 is the leftmost element of the subarray we are considering), so no work is done here. When we recursively quicksort elements to the right of 3, the elements 4 and 5 end up getting swapped. After both recursive calls complete, the array looks like this:



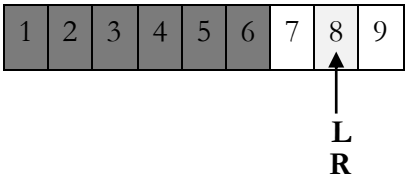
At this point, we have finished quicksorting elements to the left of our original pivot, 6. Now that all elements to the left of 6 are sorted, we can make a recursive call to quicksort the elements to the right of 6. The process is similar to before: we select the last element, 8, as the pivot, and initialize L and R.



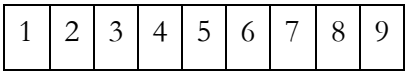
Increment L until it points to a value greater than the pivot value of 8. In this case, L ends up pointing to 9 since 9 is the first value greater than 8, but R is also pointing to 9.



Since L and R are equal, we swap 9 with the pivot.



There is only one element to the left and right of 8, which are both trivially sorted. Thus, we have finished the quicksort of the elements to the right of our initial pivot of 6. The entire quicksort algorithm is complete, and our array is fully sorted.

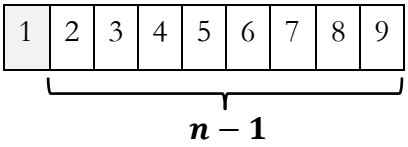


Time Complexity of Quicksort

Quicksort is a recursive algorithm. We call the ***partition()*** function, which ensures that the pivot value is placed in its correct sorted position, and that all elements to the left are smaller and all elements to the right are larger. This partitioning step takes $\boldsymbol{O(n)}$ time, since a single array pass is conducted when traversing using the left and right indices. Then, we recursively call quicksort on the elements to the left and right of the pivot.

What is the input size of these recursive calls? This actually depends on the value of the pivot!

In the worst case, either the smallest or largest element is chosen as the pivot at every step. When this happens, the pivot always leaves one side empty and the other side with all the remaining elements. For instance, if **1** were chosen as the pivot, every other element would be partitioned to the right of the pivot:



When this happens, one of the recursive calls would have an input size of **0**, and the other would have an input size of $n-1$. If either the smallest or largest element is chosen as the pivot every time, the recurrence relation for quicksort becomes

$$T(n) = \begin{cases} 1 & if\ n = 0\ or\ 1 \\ T(n-1) + n & if\ n > 1 \end{cases}$$

Here, the $T(n-1)$ comes from the recursive quicksort call with input size $n-1$, and the n comes from the partitioning step. Using Iterative Substitution Method, we can see that the worst-case time complexity of quicksort is $O(n^2)$.

II.8. Merge Sort

The basic idea of merge sorting is to divide the list to be sorted into two halves (equal halves). The two halves are then recursively sorted and merged using a ***merge()*** function. An implementation of merge sort is shown in the code below:

```
void merge_sort(int Vec[], int left, int right) {
      if (right - left < 2) {
            return;
      } // if
      int mid = left + (right - left) / 2;
      merge_sort(Vec, left, mid);
      merge_sort(Vec, mid, right);
      merge(Vec, left, mid, right);
} // merge_sort()
```

On line ***merge(Vec, left, mid, right);*** , we merge two sorted arrays together. This merging step can be done in $O(n)$ time using the following process:

1) Keep track of two indices, i and j, that refer to the first element of each of the sorted halves.
2) Initialize a separate vector of size "$size = right - left$;" to store the merged ***output***.
3) **If Vec[i] = Vec[j]** copy **Vec[i]** to the output vector and increment i.

   Otherwise, copy **Vec[j]** to the output vector and increment j.

4) Repeat until the entire output vector is filled. If one sorted half reaches the end before the other, simply append all of the remaining elements in the unfinished half into the output vector.

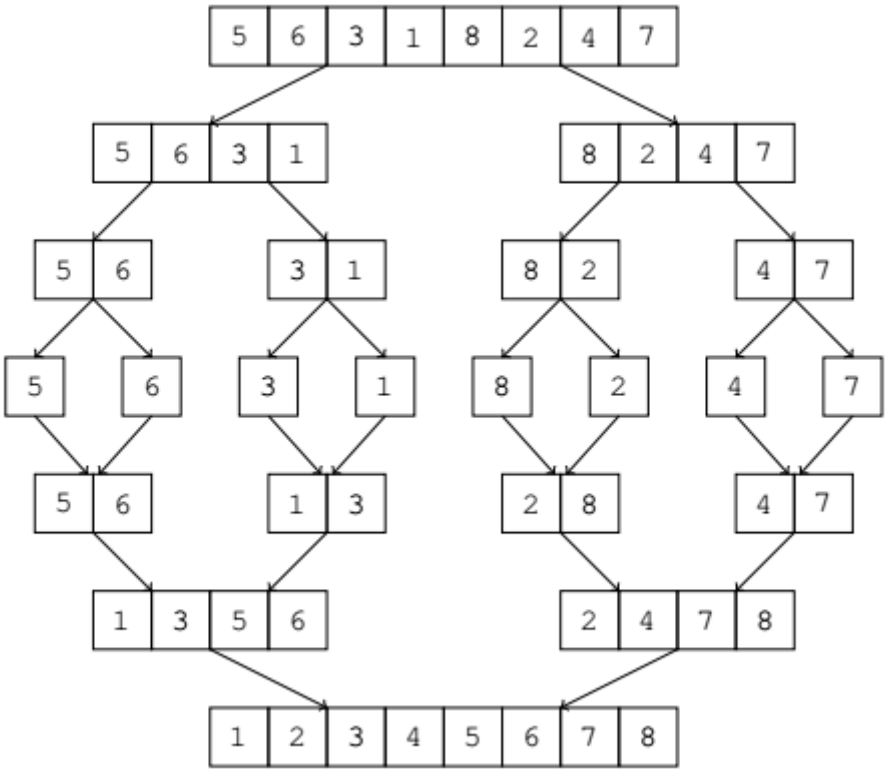
```
void merge(int Vec[], int left, int mid, int right) {
      int size = right - left;
      int output[size]; //init output vector
      for (int i = left, j = mid, k = 0; k < size; k++) {
          if (i == mid) { // if first half is complete
                  output[k] = Vec[j++];
          }
          else
              if (j == right) { // if second half is complete
                  output[k] = Vec[i++];
              }
              else {
                  if (Vec[i] <= Vec[j]) {
                        output[k] = Vec[i++];
                  }
                  else {
                        output[k] = Vec[j++];
                  }
              }
      } // for
      // copies data sorted in output[] to Vec[].
      for (int i = left; i < right; i++){
            Vec[i] = output[i-left];
      }
} // merge()
```

Example:

Let's look at what this code does using an example. Consider the following unsorted array:

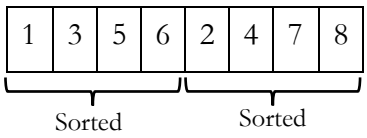
5	6	3	1	8	2	4	7

The following diagram illustrates the entire merge sort process on our initial array.



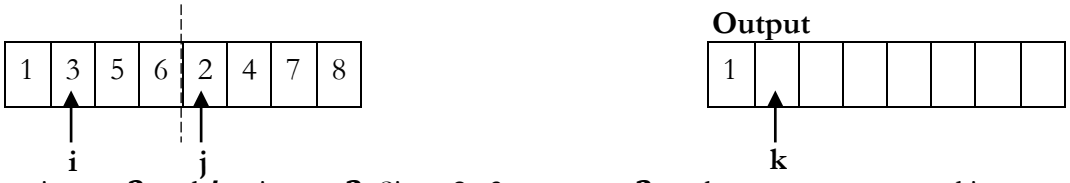
Consider the ***merge()*** function on the following unsorted array: **[5, 6, 3, 1, 8, 2, 4, 7]**.

During the merge sort process, we first recursively merge sort the two halves of the array, **[5, 6, 3, 1]** and **[8, 2, 4, 7].** This sorts the two halves of the array. After this step, the contents of the array are as follows:

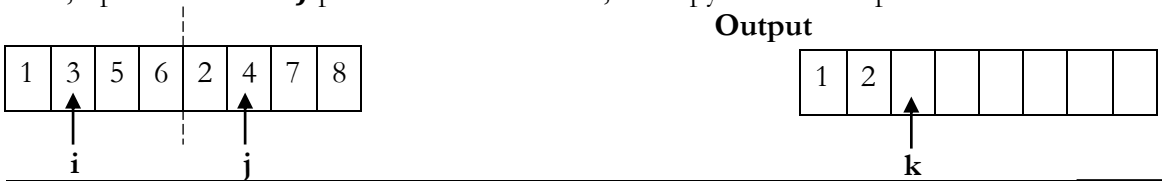


To merge the two sorted halves, we will first initialize an empty vector to store the sorted output and initialize indices ***i*** and ***j*** to refer to the first elements of the two halves. The variable ***k*** represents the next open slot in the output vector.

We then compare ***i*** and ***j*** and copy the smaller of the two values to the output vector. The index ***k*** is then incremented, along with either ***i*** or ***j*** depending on which element was copied. In this case, **1 < 2**, so we move **1** to the output vector and increment ***i***.



Now, ***i*** points to **3** and ***j*** points to **2**. Since **2<3** , we copy **2** to the output vector and increment ***j***.



Since $\mathbf{3} < \mathbf{4}$, we copy $\mathbf{3}$ to the output vector and increment $\boldsymbol{i}$.

Since $\mathbf{4} < \mathbf{5}$, we copy $\mathbf{4}$ to the output vector and increment $\boldsymbol{j}$.

Since $\mathbf{5} < \mathbf{7}$, we copy $\mathbf{5}$ to the output vector and increment $\boldsymbol{i}$.

Since $\mathbf{6} < \mathbf{7}$, we copy $\mathbf{6}$ to the output vector and increment $\boldsymbol{i}$.

Notice that $\boldsymbol{i}$ has gone off the edge of its portion of the array! As a result, we know that every element in the first half has been added to the output vector already. This means that the remaining elements in the second half are guaranteed to be larger than the elements added so far, so they can be copied to the output vector one by one.

Time Complexity of Merge sort

What is the time complexity of merge sort? If we look at the ***merge()*** function implemented above, we can see that the algorithm does a single pass through all the elements in its input vector range. Thus, the merging operation takes $\boldsymbol{O(n)}$ time. We can now express the merge sort function as a recurrence relation.

Since merge sort makes two recursive calls with input size $\mathbf{n/2}$ and calls ***merge()*** (which takes linear time), the merge sort process can be expressed using the following recurrence relation:

$$\boldsymbol{T(n)} = \begin{cases} \mathbf{1} & \boldsymbol{if\ n = 1} \\ \boldsymbol{2T(n/2) + n} & \boldsymbol{if\ n > 1} \end{cases}$$

Using Master's Theorem gives us a complexity of $\boldsymbol{O(n\ log\ n)}$.

Additionally, since merge sort initializes an output vector to store the contents of the sorted output, the auxiliary space required by the algorithm is $\boldsymbol{O(n)}$, as the size of the output vector is linear on the number of elements that need to be sorted.

Iterative version of Merge sort

The merge sort implementation we have presented is based on a top-down approach, since it uses recursion to break the input array into halves that can be sorted individually. Nevertheless, there exists an alternative method to implement merge sort iteratively, devoid of recursive invocations. The following is an implementation of this bottom-up merge sort:

```
void merge_sort(int Vec[], int left, int right) {
      for (int size = 1; size <= right - left; size *= 2) {
            for (int i = left; i <= right - size; i += 2 * size) {
                  merge(Vec, i, i + size, min(i + 2 * size, right));
            }
      }
} // merge_sort()
```

Example:

This variation of merge sort uses nested for loops to simulate the behavior of the recursive calls. First, it merges groups of two elements, then it merges groups of four elements, then it merges groups of eight elements, and so on until the entire array is sorted. To illustrate this, consider the unsorted array $[\mathbf{5}, \mathbf{6}, \mathbf{3}, \mathbf{1}, \mathbf{8}, \mathbf{2}, \mathbf{4}, \mathbf{7}]$. On the first pass, the algorithm considers elements in pairs of two and merges these pairs together:

$[\mathbf{5}, \mathbf{6}, 3, 1, 8, 2, 4, 7]$ Merge 5 and 6 in sorted order.
$[5, 6, \mathbf{1}, \mathbf{3}, 8, 2, 4, 7]$ Merge 3 and 1 in sorted order.
$[5, 6, 1, 3, \mathbf{2}, \mathbf{8}, 4, 7]$ Merge 8 and 2 in sorted order.
$[5, 6, 1, 3, 2, 8, \mathbf{4}, \mathbf{7}]$ Merge 4 and 7 in sorted order.

On the second pass, size increases to two, and the algorithm merges elements in groups of four:

$[\mathbf{1}, \mathbf{3}, \mathbf{5}, \mathbf{6}, 2, 8, 4, 7]$ Merge 5, 6, 1, and 3 in sorted order.
$[1, 3, 5, 6, \mathbf{2}, \mathbf{4}, \mathbf{7}, \mathbf{8}]$ Merge 2, 8, 4, and 7 in sorted order.

On the third and final pass, size gets increased to three, and the algorithm merges elements in groups of eight:

$[\mathbf{1}, \mathbf{2}, \mathbf{3}, \mathbf{4}, \mathbf{5}, \mathbf{6}, \mathbf{7}, \mathbf{8}]$ Merge 1, 3, 5, 6, 2, 4, 7, and 8 in sorted order.

II.9. Analysis of Comparison Sorts

All of the sorting algorithms we have discussed so far are comparison-based sorting algorithms. These sorting algorithms sort a list (array) by comparing elements with each other using a comparison operator.

Algorithm	Best Time	Average Time	Worst Time	Auxiliary Space
Bubble Sort	$\Omega(n)$	$\Theta(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$
Selection Sort	$\Omega(n^2)$	$\Theta(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$
Insertion Sort	$\Omega(n)$	$\Theta(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$
Quick Sort	$\Omega(n \log n)$	$\Theta(n \log n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$
Merge Sort	$\Omega(n \log n)$	$\Theta(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n)$