Chapter 2

Sorting algorithms

2.1 Introduction

Sorting algorithms are one of the fundamental pillars of computer science and play a vital role in the efficient processing and manipulation of data. Sorting, which is the process of rearranging the elements of a collection in a particular order, is a commonly used operation in many areas of computer science, such as information retrieval, data analysis, databases, and many others. In this chapter, we will explore a range of sophisticated and efficient techniques for organizing data sets in an orderly manner. We will cover classic sorting algorithms, such as selection sort, insertion sort, bubble sort, as well as more advanced methods such as quicksort, mergesort. We will examine the differences between the methods in terms of efficiency and complexity to enable readers to make informed choices about selecting the best method based on the specific context of the problem at hand.

2.2 Presentation

EitherHASa painting ofnelements. Sorting involves rearranging the elements ofHAS so that the elements ofHASare ordered. The order of the elements can be ascending or descending, alphabetical or numeric, or even custom, depending on the context of the problem to be solved and the nature of the data to be sorted. For example, in the case of an array of integers, we may want to sort the elements in ascending order, while in the case of an array of strings, we may want to sort the elements in alphabetical order. All sorting algorithms can adapt to any type of data, provided that the order of the elements is defined (the elements must be comparable).

2.3 Bubble Sort

Bubble sort is one of the simplest and most intuitive sorting algorithms. It is based on the idea of traversing the array multiple times, comparing adjacent elements and swapping them if they are out of order. The algorithm stops when the array is traversed without any swaps being performed. Bubble sort is a stable sorting algorithm, meaning that the order of equal elements is not changed. Bubble sort is an in-place sorting algorithm, meaning that it does not require additional memory, as it performs the comparisons and swaps directly in the input array.

2.3.1 Example

Consider the tableA = [5, 1, 4, 2, 8]to sort in ascending order. Bubble sort proceeds as follows:

— First pass:we traverse the array and compare adjacent elements. If an element is greater than its successor, we swap them. Note here that the comparison operation is chosen based on the desired sort order. If we want to sort the elements in ascending order or even sort another type of data with a more complex order, only the order test (the comparison operation) needs to be changed, the rest of the algorithm remains unchanged. In our example, we compare 5And1, 5is greater than1,so the elements are not in order, so we will exchange them. The table becomes:

$$A = [1, 5, 4, 2, 8].$$

We then compare5And4, 5is greater than4,the elements are not in order, we exchange them and the table becomes:

$$A = [1, 4, 5, 2, 8].$$

The same thing happens for the elements5And2,we exchange them and the table becomes:

$$A = [1, 4, 2, 5, 8].$$

For the elements5And8, 5is smaller than8,the elements are in order, and so we do nothing. Note that at the end of the first pass, the largest element of the array is placed at the end of the latter. Consequently, at each pass, we can reduce the size of the array to be scanned by1.That is to say that we can browse the first$n-1$elements instead of$n$for the second passage, the first$n-2$

elements for the third pass, and so on.

— Second pass:we go through then −1first elements of the array and compare adjacent elements. If an element is greater than its successor, they are swapped. In our example, we compare1And4,and the elements are not exchanged because they are in order. Then we compare4And2, 4is greater than2,we exchange them and the table becomes:

$$A = [1, 2, 4, 5, 8].$$

We then compare4And5, 4is smaller than5,the elements are in order and the array remains unchanged. As we explained earlier, we do not need to compare the last element of the array because it is already in order. The same applies for the penultimate element, in the next pass.

— Third passage:we go through then −2first elements of the array and we compare the adjacent elements. In this case, we will observe that all then−2 elements are already in order, because no swaps were made during this pass. So the algorithm stops here.

2.3.2 Implementation

Below we show an implementation of the bubble sort algorithm in C:

```
1  intbubble_sort(int*HAS,intn) {
2      inti, j, tmp;
3      for(i = 0; i < n - 1; i++) {
4          ordered = 1;
5          for(j = 0; j < n - i - 1; j++) {
6              if(A[j] > A[j + 1]) {
7                  tmp = A[j];
8                  A[j] = A[j + 1]; A[j + 1]
9                  = tmp;
10                 ordered = 0;
11             }
12         }
13         if(ordered) {
14             return;
15         }
16     }
17 }
```

Algorithm 2.1 – Bubble Sort – Iterative Version

The previous version is an iterative implementation of the bubble sort algorithm. We can also implement this algorithm recursively, as follows:

```
1  intbubble_sort_rec(int*HAS,intn) {
2      inti, tmp;
3      if(n != 1) {
4          ordered = 1;
5          for(i = 0; i < n - 1; i++) {
6              if(A[i] > A[i + 1]) {
7                  tmp = A[i];
8                  A[i] = A[i + 1]; A[i + 1]
9                  = tmp;
10                 ordered =      0;
11             }
12         }
13         if(!ordered)          {
14             bubble_sort_rec(A, n - 1);
15         }
16     }
17 }
```

Algorithm 2.2 – Bubble Sort – Recursive Version

2.3.3 Complexity

The basic operation we are interested in in the bubble sort algorithm is the comparison between two adjacent elements. The algorithm has two nested loops, the first loop ensures that the algorithm makes multiple passes over the array (at mostn −1passages), and the second loop ensures that the algorithm iterates through the elements of the array. The number of elements iterated through in the second loop depends on the current pass, i.e. in the first pass, we iterate through all the elements of the array (we don −1comparisons), in the second passage, we go through then −1first elements of the array, and so on. On each iteration in the second loop, only one comparison is performed. The algorithm stops when all elements are in the order in the last pass. But in the worst case (the array is sorted in reverse order), we performn−1passages. The number of comparisons made in the worst case is therefore:

$$T_{bubble}(n) = (n-1) + (n-2) + \cdots + 1 = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2} \quad (2.1)$$

The algorithm therefore has a quadratic complexity, that is to say$O(n^2)$.

We can apply the same reasoning for the recursive version of the algorithm, noting that the number of comparisons performed for an array of sizenEast n−1 comparison. Then we call the function recursively with an array of size

n−1and so on. The recursive process stops when the array is sorted, or at worst, when a recursive call is made with an array of size1 (that is to say aftern−1recursive calls).

2.4 Sorting by selection

Selection sort is a sorting algorithm that consists of finding the smallest element in the array and placing it in the first position (in the case of ascending sort), then finding the second smallest element and placing it in the second position, and so on. The selection sort algorithm is also an in-place algorithm, meaning that it does not require additional memory to perform the sort. The algorithm stops when all the elements are in order.

2.4.1 Example

We will illustrate how the selection sort algorithm works using the following example:

$$A = [5, 1, 4, 2, 8].$$

— During the first pass, we look for the smallest element of the array, which is1,and we exchange it with the element at the first position of the array. The array becomes:

$$A = [1, 5, 4, 2, 8].$$

— On the second pass, we look for the smallest element in the array, but this time we ignore the first element. The smallest element from the second position is2,we exchange it with the element at the second position of the array. The array becomes:

$$A = [1, 2, 4, 5, 8].$$

Note that at this stage the array is already sorted, but unlike the bubble sort algorithm, the selection sort algorithm has no way of knowing whether the array is sorted or not, so it continues to perform subsequent passes, eliminating the k−1 first elements of the array at each pass, wherekis the number of the current pass. The table will remain unchanged until the last pass.

2.4.2 Implementation

Below we show an implementation of the iterative version of the selection sort algorithm in C language:

```
1  intselection_sort(int*HAS,intn) {
2      inti, j, ind_min , tmp; for(i = 0; i < n - 1;
3      i++) {
4          ind_min = i;
5          for(j = i + 1; j < n; j++) {
6              if(A[j] < A[ind_min ]) {
7                  ind_min = j;
8              }
9          }
10         if(ind_min != i) {
11             tmp =    A[i];
12             A[i] =       A[ind_min ];
13             A[ind_min]       = tmp;
14         }
15     }
16 }
```

Algorithm 2.3 – Selection sort – iterative version

Below we show an implementation of the recursive version of the selection sort algorithm in C language:

```
1  intselection_sort_rec(int*HAS,intn) {
2      inti, ind_min , tmp; if(n !=
3      1) {
4          ind_min = 0;
5          for(i = 1; i < n; i++) {
6              if(A[i] < A[ind_min ]) {
7                  ind_min = i;
8              }
9          }
10         if(ind_min != 0) {
11             tmp =    A[0];
12             A[0] =      A[ind_min ];
13             A[ind_min]       = tmp;
14         }
15         selection_sort_rec(A + 1, n - 1);
16     }
17 }
```

Algorithm 2.4 – Selection Sort – Recursive Version

2.4.3 Complexity

The basic operation we are interested in here is also comparison. The algorithm has two nested loops, the first loop ensures that the algorithm performsn −1passes over the array, and the second loop iterates through the elements of the array starting at positioni+1 (Oriis the number of the current pass) to find the minimum. The number of comparisons made in the second loop depends on the current pass, i.e. in the first pass, we go throughn −1elements, in the second passage, we go throughn −2 elements, and so on. The number of comparisons made is therefore:

$$T_{selection}(n) = (n-1) + (n-2) + \cdots + 1 = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2} \quad (2.2)$$

The algorithm therefore has a quadratic complexity, that is to sayO(n^2).

The same reasoning can be applied to the recursive version of the algorithm in a similar way to the bubble sort algorithm. We still arrive at the same quadratic complexity.

2.4.4 Comparison with bubble sort

When considering comparison as the basic operation, the selection sort algorithm always performs the same number of comparisons regardless of the order of the array elements. Whereas the bubble sort algorithm might perform fewer comparisons if it finds that the array is sorted at the end of a pass. However, the selection sort algorithm might be more efficient than the bubble sort algorithm in some cases, because it performs fewer swaps than the bubble sort algorithm.

Change of perspective

When considering the exchange as the basic operation, the selection sort algorithm performs a maximum of one exchange per pass. Since the number of passes isn −1,the complexity of the selection sort algorithm in this context isO(n) (linear complexity). While the bubble sort algorithm performs at worst one swap for each comparison (worst case - the array is sorted backwards), and the number of comparisons is the same as the number of swaps, the complexity of the bubble sort algorithm in this context isO(n^2) (quadratic complexity). This behavior explains the difference in performance between the two algorithms.

— The selection sort algorithm is more efficient than the bubble sort algorithm if it is run on a hardware architecture where swapping is more expensive than

comparison, especially on processors with small registers or small cache memory.

— The bubble sort algorithm is more efficient than the selection sort algorithm if it is executed on a hardware architecture where comparison is more expensive than swapping. The comparison operation when used in a condition will generate a conditional jump, which is an expensive operation in terms of execution time, especially on processors with deep pipelines or a poorly performing branch predictor.

The difference between the two algorithms may also depend on the nature of the data to be sorted. If the data is already (almost) sorted, the bubble sort algorithm will be more efficient than the selection sort algorithm, because it performs fewer passes over the array. The algorithm to use between the two therefore depends on the nature of the data to be sorted and the hardware architecture on which the algorithm will be executed.

2.5 Insertion Sort

Insertion sort algorithm is a sorting algorithm that inserts each element of the array into its place in a sorted array. The algorithm starts by considering that the first element of the array is sorted, then it inserts the second element into its place in the sorted array, then it inserts the third element into its place in the sorted array, and so on until all the elements of the array are inserted into their place. Insertion sort algorithm is a stable algorithm (the order of equal elements is preserved) and in-place algorithm (the algorithm does not require additional memory). This algorithm relies on the complexity of insertion into a sorted array, which is sub-linear (logarithmic) to improve the complexity of the sorting algorithm.

2.5.1 Example

Below we show an example of running the insertion sort algorithm on the following 5-element array:

$$A = [\{5\}, 2, 4, 6, 1]$$

The tableHASis divided into two parts, the first part is the sorted array, and the second part is the remainder of the array. The size of the sorted array is initially 1, and the size of the remainder of the array is initiallyn −1 (Ornis the size of the array). The algorithm starts by inserting the second element of the array into the sorted array, then it inserts the third element of the array into the sorted array, and so on until all elements

of the table are inserted in their place. We note the sorted part of the table by putting it in braces.

— Passage 1:on the first pass, the sorted array is [{5}]and the rest of the table is [2, 4, 6, 1].The first step is to insert the second element of the array (the value2) in the sorted table we get as result:

$$A = [\{2, 5\}, 4, 6, 1]$$

.

— Passage 2:on the second pass, the sorted array is [{2, 5}]and the rest of the table is [4, 6, 1].We will insert the value4in the sorted table we get as result:

$$A = [\{2, 4, 5\}, 6, 1]$$

.

— Passage 3:on the third pass we will insert the value6in the sorted table we get as result:

$$A = [\{2, 4, 5, 6\}, 1]$$

.

— Passage 4:The fourth pass consists of inserting the last element of the array (the value 1)in the sorted part of the table, which gives us the following sorted table:

$$A = [\{1, 2, 4, 5, 6\}]$$

.

2.5.2 Implementation

Before implementing the insertion sort algorithm, it is very important to implement and understand the process of inserting into a sorted array. The naive version of inserting into a sorted array is to traverse the array looking for the insertion position (the first element that is greater than the element to be inserted in the case of ascending sort), then shift all the elements in the array from the insertion position one square to the right, and then insert the element in its place. This version of insertion is implemented in C language below:

Inserting into a sorted table

```
1  void insert(int* array ,int size ,int element) {
2      int i = 0;
3      while(i < size && array[i] < element) i++; for(int j = size; j >
4      i; j--) {
5          array[j] = array[j - 1];
6      }
7      array[i] = element;
8  }
```

Algorithm 2.5 – Insertion into a sorted array - Linear complexity

Note that the functioninserttakes an array of integers as a parameterarray,the size of the table sizeand the element to be insertedelement.The functioninsertreturn the tablearray with the elementelementinserted in its place assuming that the tablearrayis followed by an empty box that can be used to ensure that the offset of the array elements does not exceed its size.

A more efficient version of inserting into a sorted array can be implemented by using dichotomous search to find the insertion position of the element. Dichotomous search is a search that consists of dividing the array into two parts, then finding in which part the element to be searched is located, the array is then divided into two parts each time as follows:

— If the element to be inserted is larger than the element in the middle of the array, then the element must be inserted into the second part of the array.

— If the element to be inserted is smaller than the element in the middle of the array, then the element must be inserted into the first part of the array.

The same process is repeated until the insertion position of the element is found, or we end up with an array of size 0.

The version of inserting into a sorted array using dichotomous search is implemented in C language below:

```
1  void insert(int* array ,int size ,int element) {
2      int i = 0, j = size - 1, k; while(i <= j) {
3
4          k = (i + j) / 2;
5          if(array[k] < element) i = k + 1; else j = k - 1;
6
7      }
8      for(k = size - 1; k > i; k--) {
9          array[k] = array[k - 1];
```

```
10      }
11      array[i] = element;
12  }
```

Algorithm 2.6 – Insertion into a sorted array - Logarithmic complexity

The insertion position finding process can also be implemented recursively, yielding the following version:

```
1   intsearch(int*array ,inti,intI,intelement) {
2       if(i > j)returni; intk = (i +
3       j) / 2;
4       if(array[k] < element)returnsearch(array, k + 1, j, element); else returnsearch(array, i,
5       k - 1, element);
6   }
7
8   voidinsert(int*array ,intsize ,intelement) {
9       inti = search(array, 0, size - 1, element); for(intk = size - 1; k
10      > i; k--) {
11          array[k] = array[k - 1];
12      }
13      array[i] = element;
14  }
```

Algorithm 2.7 – Insertion into a sorted array - Recursive version

Once we have implemented the insert function into a sorted array, we can now move on to insertion sort. We will iterate over the array to be sorted, inserting each element into the sorted array, which gives us the following implementation:

```
1   voidinsertion_sort(int*array ,intsize) {
2       for(inti = 1; i < size; i++)
3           insert(array, i, array[i]);
4   }
```

Algorithm 2.8 – Insertion sort

2.5.3 Complexity

We consider comparison as the basic operation of the insertion sort algorithm. Its complexity mainly depends on the complexity of the insertion function in a sorted array. In the case of using dichotomous search to find the insertion position of the element, the complexity of the insertion function is logarithmic, since dichotomous search divides the array into two parts each time. In other words, the number of comparisons performed to find the

insertion position in a sorted array of sizenis limited by the number of times it can be dividednby 2, which gives the worst-case logarithmic complexity$O(\log_2 n)$.

The insertion is donentimes in the case of insertion sort, which gives a complexity of$O(n \log n)$in the worst case. This complexity is a bit pessimistic, because the size of the array into which the elements are inserted starts at 1 and increases each time, which means that the number of comparisons performed to insert the elements is given by the following formula:

$$T_{inssort}(n) = \log_2 1 + \log_2 2 + \log_2 3 + \cdots + \log_2 n \tag{2.3}$$

But the result of this sum is always less than the following sum:

$$\log_2 n + \log_2 n + \log_2 n + \cdots + \log_2 n = n \log_2 n \tag{2.4}$$

Hence the complexity$O(n \log_2 n)$.

2.5.4 Comparison with other sorting algorithms

Comparing sorting algorithms is a difficult task because several operations can be considered as basic operations. Very often, the number of comparisons is considered as the basic operation, and in this case, it seems that insertion sort is more efficient than selection sort and bubble sort. However, insertion sort performs more memory operations than selection sort. Indeed, in the worst case, when memory operations are considered as basic operations, insertion sort has a complexity of$O(n_2)$, which is the same complexity as bubble sort, while selection sort has a complexity of$O(n)$,which is more efficient than the other two algorithms. In addition to this, the insertion sort algorithm is adaptive, i.e. if the array is almost sorted, then the number of operations performed is very small.

In conclusion, the most efficient algorithm among the three seen so far depends on the situation, the questions to ask are:

— Is the table partially sorted?

— Is a memory operation more expensive than a comparison operation?

— Is the branch predictor effective?

— Is the number of registers limited?

— Is the cache memory sufficient?

2.6 Merge Sort

Merge sort is a recursive algorithm that divides the array to be sorted into two parts, sorts each part, and then merges them to obtain the sorted array. The merge function takes two sorted arrays and merges them into a single array that is always sorted. The merge sort algorithm is an external sorting algorithm, that is, it must use external memory space to store the data to be sorted. The recursive calls stop when the array to be sorted contains only one element, because an array of a single element is always sorted.

Merge sort is a somewhat advanced sorting algorithm and it is very efficient when sorting a large array. It is also very efficient when sorting data stored on an external medium, because it copies a part of the data to be sorted into the main memory each time, which avoids repetitive accesses to external memories which are very slow.

2.6.1 Example

We will illustrate how the merge sort algorithm works using an example. We will sort the arrayA = [5, 3, 8, 6, 2, 7, 1, 9].The merge sort process is illustrated in Figure 2.1. We have specifically chosen a size chart$2_k$for ease of explanation. However, the merge sort algorithm also works for arrays of size not a power of 2. In the upper part of the figure2.1, the array to be sorted is divided into two equal parts each time until we get arrays of size 1 (which is sorted by definition). In the lower part, the sorted arrays are merged until we get the final sorted array.

2.6.2 Implementation

The merge sort algorithm is implemented using a recursive function that takes as parameter the array to be sorted with its size. The recursive function divides the array into two parts, then recursively calls the sort function on each part. When the size of the array is equal to 1, the function returns the array. When the two sorted arrays are returned, another function is called to merge them into a single array that is still sorted. We will first start with the implementation of the merge function that takes as parameter two successive sorted sub-arrays and merges them into a single sorted array that occupies the same space as the two merged arrays. The function must use an external memory space to store the merged array, then copy it into the array

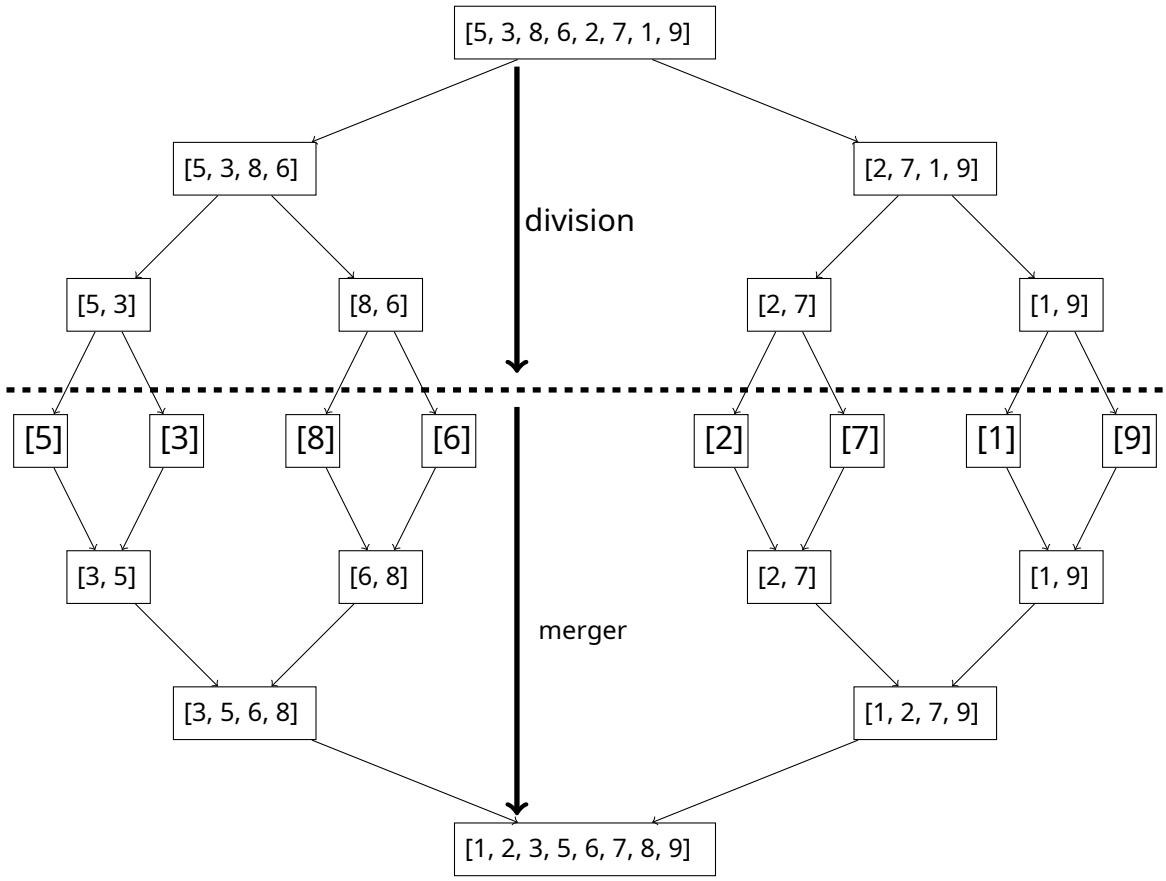

Figure 2.1 –Merge Sort

initial. The implementation of the merge function is shown below.

```
1   void merge(int *first ,int *second,int size_first ,int size_second) {
2       int *merged = malloc ((size_first + size_second) *sizeof(int)); int i = 0, j = 0, k = 0;
3
4       while(i < size_first && j < size_second) {
5           if(first[i] < second[j]) {
6               merged[k]     =   first[i];
7               i++;
8           }   else    {
9               merged[k]     =   second[j];
10              j++;
11          }
12          k++;
13      }
14      while(i < size_first) {
15          merged[k] = first[i];
16          i++;
17          k++;
18      }
```

```
19      while(j < size_second) {
20          merged[k] = second[j];
21          j++;
22          k++;
23      }
24      for(i = 0; i < size_first + size_second; i++) {
25          first[i] = merged[i];
26      }
27      free(merged);
28  }
```

Algorithm 2.9 – Merge two sorted arrays

We will now implement the merge sort function. This recursive function takes as parameter the array to sort and its size. If the size of the array is equal to 1, then the function returns the same array. Otherwise, the function divides the array into two parts, then recursively calls the sort function on each part. When both sorted arrays are returned, we use the merge function in the algorithm2.9to merge them. The implementation of the merge sort function is shown below:

```
1   void merge_sort(int *array ,int size) {
2       if(size != 1) {
3           int size_first = size / 2;
4           int size_second = size - size_first; int *first =
5           array;
6           int *second = array + size_first;
7           merge_sort(first,           size_first);
8           merge_sort(second ,       size_second);
9           merge(first, second,           size_first , size_second);
10      }
11  }
```

Algorithm 2.10 – Merge sort

2.6.3 Complexity

As with other algorithms, we consider that the comparison operation is the most expensive operation. We will therefore estimate the complexity of the merge sort algorithm by counting the number of comparisons performed. The complexity of the merge sort algorithm depends on (1) the complexity of the merge function and (2) the number of recursive calls, i.e. the depth of the recursion. The complexity of the merge function is linear, because it traverses the two arrays to be merged only once. The number of recursive calls cannot exceed the number of times the array size can be divided by 2 until we obtain 1, which is equal to$\log_2 n$Ornis the size of the initial array. Since at each level of the recursion, each element is compared a

times at most (ncomparisons at each level), and that the depth of the recursion is$\log_2 n$,the complexity of the merge sort algorithm is of the order of$O(n \cdot \log_2 n)$.

If we are interested in memory operations, the merge sort algorithm always has a complexity of the order of$O(n \cdot \log_2 n)$,because at each level of recursion,nelements are copied into an external array, then copied back into the initial array (T(n) =2n).The depth of the recursion is$\log_2 n$,so the complexity of the merge sort algorithm is of the order of $O(n \cdot \log_2 n)$.

2.6.4 Comparison with other algorithms

The merge sort algorithm is more efficient than the other sorting algorithms we have seen so far. In fact, the complexity of the merge sort algorithm is of the order of$O(n \cdot \log_2 n)$,whatever the operation considered. This is the optimal complexity for comparison sort algorithms. The merge sort algorithm is therefore more efficient than bubble sort, insertion sort and selection sort algorithms, particularly when the size of the table to be sorted is large.

2.7 Quick sort

Quicksort is a comparison sorting algorithm that uses the technique ofdivide and conquer.Like merge sort, it is based on the technique ofpartitioning which involves dividing the array into two parts and then sorting each part separately. The quicksort algorithm is a recursive algorithm that uses a partitioning function to divide the array into two parts. The partitioning function chooses an element from the array calledpivot,then places all elements smaller than the pivot to its left and all elements larger than the pivot to its right. The quicksort algorithm is also an in-place algorithm, because it does not require an external array to perform the sort.

2.7.1 Example

We will illustrate how the quicksort algorithm works in the following table:

$$A = [5, 3, 7, 2, 1, 4, 6]$$

First we will choose a pivot. This can be chosen in different ways, but the most common choice is to choose the last element of the array. In our

example, the pivot is equal to6.We will then traverse the array from left to right, and place all elements smaller than the pivot to its left. When we find an element smaller than the pivot, we swap it with the first element larger than the pivot. When we have traversed the entire array, we swap the pivot with the first element larger than it. The array then becomes:

$$A = [\{5, 3, 2, 1, 4\},6, \{7\}]$$

Then, we apply the same procedure on the two parts of the array separated by the pivot. The right part of the array is already sorted, because it contains only one element. We therefore apply the procedure on the left part of the array. The last element of the sub-array is chosen as the pivot, its value is4,after this step, the table becomes:

$$A = [\{3, 2, 1\},4, \{5\}, 6, 7]$$

We apply the same procedure to the sub-table of [3, 2, 1],the last element is chosen as pivot, its value is1,after this step, the table becomes:

$$A = [1, \{2, 3\}, 4, 5, 6, 7]$$

In the last step, we apply the same procedure on the sub-table of [2, 3],it is already sorted, but to finish the example we will still choose the last element as pivot, its value is3,after this step, the table becomes:

$$A = [1, \{2\},3,4, 5, 6, 7]$$

At the end of this last step, the table is sorted.

2.7.2 Implementation

The implementation of the quicksort algorithm is given below:

```
1  void quick_sort(int *array ,int size) {
2      if (size > 1) {
3          int pivot = array[size - 1]; int i = 0;
4
5          int j = size - 2; while (i
6          <= j) {
7              if (array[i] > pivot && array[j] < pivot) {
8                  int tmp = array[i];
```

```
 9                 array[i]       =  array[j];
10                 array[j]       =  tmp;
11                 i++;
12                 j--;
13             }else if(array[i] <= pivot) {
14                 i++;
15             }else if(array[j] >= pivot) {
16                 j--;
17             }
18         }
19         inttmp = array[i]; array[i] =
20         array[size - 1]; array[size - 1] = tmp;
21         quick_sort(array, i);
22
23         quick_sort(array + i + 1, size - i - 1);
24       }
25   }
```

2.7.3 Complexity

Let us consider the comparison operation as the basic operation. The complexity of the quicksort algorithm in the worst case (when the array is already sorted) is of the order of $O(n_2)$, because at each level of the recursion, each element is compared to the pivot once at most (n − icomparisons such asiis the level of recursion). The depth of recursion isn,so the complexity of the quicksort algorithm is of the order of $O(n_2)$. The complexity of the quicksort algorithm in the best case (when the pivot is always in the middle of the array) is of the order of$\Omega(n \cdot \log_2 n)$,because at each level of the recursion, each element is compared to the pivot once at most (n−icomparisons such asiis the level of recursion). The depth of recursion is$\log_2 n$,so the best-case complexity of the quicksort algorithm is of the order of$\Omega(n \cdot \log_2 n)$.The complexity of the quicksort algorithm on average is of the order of$n \cdot \log_2 n$,because on random tables of sizen,it is very unlikely to have an already sorted table.

2.7.4 Comparison with merge sort

Although the worst-case complexity of the quicksort algorithm is of the order of$O(n_2)$In the worst case, this algorithm is on average more efficient than the merge sort algorithm. Indeed, the average complexity of the quick sort algorithm is of the order of $O(n \cdot \log_2 n)$,the same as the merge sort algorithm. Furthermore, the quick sort algorithm is an in-place algorithm, while the merge sort algorithm requires an external array to perform the sorting. Furthermore, in the worst case for the quick sort algorithm, no copy operation is performed, and the comparison with the pivot gives

always a negative result, which allows the branch predictor to adapt and optimize the performance of the algorithm.

2.8 Conclusion

In this chapter, we have presented several sorting algorithms. We started with the simplest algorithms to understand and implement, bubble sort and selection sort. Both of these algorithms have a complexity of the order of$O(n_2)$, and are therefore very inefficient on large arrays. We then presented insertion sort, which has a complexity of the order of$O(n \cdot \log_2 n)$when it comes to comparisons, but of the order of$O(n_2)$if we consider memory operations. We then presented merge sort, which has a complexity of the order of$O(n \cdot \log_2 n)$,which is much faster than previous algorithms on large arrays. Finally, the quicksort algorithm was presented, it has a complexity at worst of the order of$O(n_2)$, but in practice it is faster than merge sort, because it is in place and its average complexity is of the order of$n \cdot \log_2 n$.

The choice of which sorting algorithm to use depends on several factors, such as the size of the array to be sorted, the complexity of the algorithm, the nature of the data to be sorted, the amount of memory available, etc. In practice, quick sort is the most widely used sorting algorithm, including in the standard libraries of programming languages. However, in some situations, other sorting algorithms may be more efficient.