1 The Problem

Sorting is the problem of reorganizing a sequence of elements according to a predefined order. Typical example is that of organizing a sequence of numbers in increasing or decreasing order. In the context of this course we will limit our discussion to the so-called Internal Sorting: the process of reorganizing a sequence of elements according to a given order will be realized by keeping all data in main memory during the sorting process. Other methods, that we will not see, make use of external storage (e.g., disk) during the sorting process, and they are called External Sorting.

Let us assume also in the rest of this section that the elements are to be sorted in ascending order, i.e., the final list of elements produced should be in increasing order.

Some of the properties commonly used to characterize a sorting algorithm are:

- **Stability**: if the list of elements contains repetitions, then relative order of the repeated elements is not modified by the sorting process.
- **Efficiency**: number of comparisons that are needed to produce the sorted list;
- **Passes**: the number of times the list of elements has to be traversed to produce the sorting.

In this context we will distinguish three categories of sorting (where \( n \) is the number of elements present in the sequence):

1. **Simple Sort**: these techniques are characterized by a worst case time complexity of \( O(n^2) \);
2. **Advanced Sort**: these techniques are characterized by a worst case time complexity of \( O(n \log_2 n) \);
3. **Radix Sort**: a special scheme which allows sorting of numbers with complexity \( O(k \times n) \), where \( k \) is the maximum number of digits present in the numbers to be sorted.

The input to a sorting algorithm is represented by a sequence of elements. In most of the following algorithms we will consider, unless explicitly stated, the elements stored in an array. Also, we will assume the sorted sequence to be stored in the same array.

2 Simple Sort

We will consider three possible simple sort schemes.

2.1 Selection Sort

This is the simplest approach to sorting a sequence of elements. Given a sequence of \( n \) numbers to be sorted, we start by computing the minimum value in the sequence, and we store it in the first position of the array (i.e., we swap the element in the first position with the minimum element). This will guarantee that the minimum element is in the correct position. At this point we can repeat the process by searching the minimum in the the
whole array but the first position. This new minimum will be placed in the second position of the array. And so on. The process terminates when we have processed all the first \( n - 1 \) locations of the array.

Thus, at every phase of the selection sort algorithm the array is partitioned in two parts; the first part is already correctly sorted, while the second part contains the elements that have not been sorted yet. The “wall” between the two sections is moved on position forward at the end of each phase. This is shown in Figure 1.

Consider the following example: we have an array containing the values

\[
390, 205, 182, 45, 235
\]

At the first step we will detect the minimum (45) and we will move it in the first position of the array (shifting 390 in the place previously used by 45)

\[
45, 205, 182, 390, 235
\]

at this point we can forget about the first position of the array and repeat the process with the remaining four positions; the new minimum is 182 which is moved in the second position of the array

\[
45, 182, 205, 390, 235
\]

we repeat the process on the last three positions of the array. The new minimum is 205 which is already in the correct position. We repeat the process on the last two elements of the array. The new minimum is 235 which should be moved to the fourth position of the array

\[
45, 182, 205, 235, 390
\]

The following code will implement the selection sort:

```c
void SelectSort(int array[], int size)
{
    int i, j, min;
    int temp;

    if (size > 1)
    {
        for (i = 0; i < size-1; i++)
        {
            min = i;
            for (j = i+1; j < size; j++)
                if (array[min] > array[j])
                    min = j;

            temp = array[min];
            array[min] = array[i];
            array[i] = temp;
        }
    }
}
```

If we consider the algorithm, we can see that the number of steps performed is

\[
(n - 1) + (n - 2) + \ldots + 1 = \frac{(n - 1)n}{2}
\]
This allows to conclude that the algorithm has a time complexity of $O(n^2)$ in the worst case.

Selection sort is the representative of a class of sorting algorithms that go under the name of Selection Sorts. Another representative of this class is the heap sort.

### 2.2 Bubble Sort

Bubble sort is based on repeated scans of the array. At every scan, successive pairs of elements are compared and swapped if the first element is greater than the second. The scan starts from one end of the array and terminates at the other end.

As in the case of selection sort, the list of elements is divided in two parts: the top part contains the elements that have already been sorted, while the bottom part contains elements that have not been sorted yet. Each scan will push the maximum element of the unsorted part to the top of the unsorted part. This is illustrated in Figure 2.

![Bubble Sort Diagram](image)

**Figure 2: Bubble Sort**

For example, during the first scan, if the initial list is

390, 205, 182, 45, 235

then first we will compare 390, 205 and exchange them (as 390 is greater than 205)

205, 390, 182, 45, 235

then we will compare the next pair (390, 182) and swap them again

205, 182, 390, 45, 235

next we will compare 390, 45 and swap them

205, 182, 45, 390, 235

and finally we will compare 390, 235 and swap them

205, 182, 45, 235, 390

After one scan we are guaranteed that the maximum element is in the last position of the array. Thus the successive scan can ignore the position (stop in position $n - 1$). The process is repeated until a scan is performed in which no elements are swapped (i.e., all the elements are already in the correct place). We are guaranteed that this will happen in at most $n - 1$ scans.

The following is the code for this sorting:

```c
void BubbleSort(int array[], int size)
{
    int i, j, temp;
    int sorted;

    if (size > 1)
    {
        sorted = 0;
        while (!(sorted) && (size > 1))
        {
            sorted = 1;
            for (i = 0; i < size-1; i++)
                if (array[i] > array[i+1])
                {
                    temp = array[i];
                    array[i] = array[i+1];
                    array[i+1] = temp;
                    sorted = 0;
                }
        }
    }
}
```

3
\begin{verbatim}
{
    temp = array[i];
    array[i] = array[i+1];
    array[i+1] = temp;
    sorted = 0;
}
size--;
}

This algorithm is going to perform the following number of steps:

\[(n - 1) + (n - 2) + \ldots + (n - k) = \frac{(2n - k - 1)(k)}{2}\]

where \(k\) is the number of executions of the outermost loop. If the elements are already sorted at the beginning, then \(k = 1\), and the complexity is proportional to \(n\). In the worst case, \(k = n - 1\) and the complexity is proportional to \(n^2\) \((O(n^2))\).

2.3 Insertion Sort

The intuition in the case of insertion sort is that of inserting an element in a sequence in such a way that the sequence is still correctly sorted. Consider the following example, we have the initial sequence

390, 205, 182, 45, 235

The last element 235, if considered by itself represents a sorted sequence (of length 1). Let us consider the previous element, 45. What we want is to insert 45 in the sequence that currently contains only 235, and such insertion should produce a sorted sequence of length 2. Since 45 is smaller than 235, then nothing special has to be done. Now the last two elements of the sequence represent a correctly sorted sequence of length 2. Let us consider the previous element, which is 182. We want to insert 182 in the sequence 45, 235. To have a properly sorted sequence, 182 has to be inserted in between the two elements. Once this is done the array looks like:

390, 205, 182, 45, 235

The last three elements represent a properly sorted sequence. Now we can insert in the sequence 45, 182, 235 the element 205, thus producing

390, 45, 182, 205, 235

Finally we want to insert 390 in the sequence 45, 182, 205, 235 and this will produce

45, 182, 205, 235, 390

The intuitive idea behind this algorithm is described in Figure 3.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{insertion_sort_diagram.png}
\caption{Insertion Sort}
\end{figure}

The main problem in this scheme is that, since we are reusing always the same array, some shifting of elements may be required to insert an element in the sequence.

The code is the following:
void InsertSort(int array[], int size)
{
    int i, j, temp;

    if (size > 1)
    {
        for (i=size-2; i >= 0; i--)
        {
            temp = array[i];
            j = i+1;
            while ((array[j] < temp) && (j <= size-1))
            {
                array[j-1] = array[j];
                j++;
            }
            array[j-1] = temp;
        }
    }
}

The complexity of this algorithm is

\[ 1 + 2 + \ldots + (n - 1) = \frac{n(n-1)}{2} \]

3 Advanced Sort

3.1 Quicksort

The quicksort scheme tries to split the problem in simpler subproblems. If we have to sort an array, then at each step one element will be selected, and the array arranged in such a way to have an organization as follows:

\[ a[0], a[1], \ldots, a[j-1], a[j], a[j+1], a[j+2], \ldots, a[n-1] \]

with the property that all the elements in the first \( j \) positions are smaller than the element in \( a[j] \), and all the elements in positions from \( j + 1 \) to \( n - 1 \) are all bigger than the element in \( a[j] \). This allows to separately sort the two partitions and produce a resulting sorted sequence. Each of the two partitions can be sorted using the same process. The element \( a[j] \) is called the pivot of the sequence. In the traditional quicksort (as developed by the inventor, C.R. Hoare), the pivot is the first element of the sequence.

For example, from the sequence

\[ 53, 59, 56, 52, 55, 58, 51, 57, 54 \]

you can partition as

\[ 51, 52, 53, 56, 55, 58, 59, 57, 54 \]

The general structure of the algorithm is as follows:

Quicksort(array, size)
    if (size == 1)
        < array already sorted >
    else
        select a[0] as pivot;
        split a[1..n] in two parts:
            b which contains all elements smaller than a[0]
            c which contains all elements bigger than a[0]
        Quicksort(b) -> sb
        Quicksort(c) -> sc
        Merge: sb + a[0] + sc
Let us consider the execution on the sequence

\[40, 15, 30, 25, 60, 10, 75, 45, 65, 35, 50, 20, 70, 55\]

At the first step we select 40 and we rearrange the sequence in such a way that we have first all the elements smaller than 40, the 40, and then all the elements greater than 40. This process can start from the two opposite extremes and move two pointers down until we find two elements that are out of place, in this case we will be able to move a pointer forward from left to the element 60, and a pointer backward from the end until the element 20. These two elements have to be swapped, thus producing:

\[40, 15, 30, 25, 20, 10, 75, 45, 65, 35, 50, 60, 70, 55\]

The process can be repeated from there, the pointer forward will stop at 75 while the pointer backward will stop at 35. A new swap is performed:

\[40, 15, 30, 25, 20, 10, 35, 45, 65, 75, 50, 60, 70, 55\]

The new scan will stop again at 45 for the forward pointer, while the backward pointer will reach 35. As the two pointer have crossed each other, we know we can stop. We finally need to insert the first element in the correct position, thus producing

\[35, 15, 30, 25, 20, 10, 40, 45, 65, 75, 50, 60, 70, 55\]

The complete code is as follows:

```c
void Quicksort(int array[], int size)
{
    RecursiveQuick(array, 0, size - 1);
}

void RecursiveQuick(int a[], int left, int right)
{
    int j, k;
    if (left < right)
    {
        j = left;
        k = right;
        if (a[left] > a[right])
            Swap(&a[left], &a[right]);

        do
        {
            do
                j++;
            while (a[j] < a[left]);

            do
                k--;
            while (a[left] < a[k]);

            if (j < k)
                Swap(&a[j], &a[k]);
        } while (k >= j);
        Swap(&a[left], &a[k]);
        RecursiveQuick(a, left, k-1);
        RecursiveQuick(a, k+1, right);
    }
}
```
The principle of MergeSort is similar to that of Quicksort. Also here we split the sequence and perform recursive sorting on the partitioned list. This is very messy to be realized using arrays. Thus we present in this case a version of Mergesort which manipulates lists of elements represented as linked lists.

The idea is the following: given the sequence

\[56, 29, 35, 42, 15, 41, 75, 21\]

at the first step we split the sequence in two and we sort the two halves separately; thus we sort

\[56, 29, 35, 42\] \text{ and } \[15, 41, 75, 21\]

This will produce

\[29, 35, 42, 56\] \text{ and } \[15, 21, 41, 75\]

The two sequence have then to be merged producing the resulting sorted list.

Let us assume that the sequence is represented as a linked list and we are passing around the pointer to the first element of the list.

The code is the following:

```c
struct node
{
    int value;
    struct node *next;
};

typedef struct node *NodePtr;

NodePtr MergeSort(NodePtr list, int size)
{
    NodePtr p1, p2, scan;
    int len;

    if (size > 1)
    {
        len = size / 2;
        p1 = list;
        scan = list;

        for (i=len; i > 1; i--)
            scan = scan->next;
        p2 = scan->next;
        scan->next = NULL;

        p1 = MergeSort(p1, len);
        p2 = MergeSort(p2, size-len);
        return (MergeLists(p1, p2));
    }
    else return (list);
}

NodePtr MergeLists(NodePtr l1, NodePtr l2)
{
    NodePtr scan, prev, top;
```
if (l1==NULL)
    return l2;
if (l2==NULL)
    return l1;

top = l1;
scan = l1;
prev = NULL;

while ((scan != NULL) && (l2 != NULL))  
{  
    NodePtr temp;
    if (scan->value > l2->value)  
    {  
        temp = l2;
        l2 = l2->next;
        if (prev != NULL)  
        {  
            prev->next = temp;
        }
        else top = temp;
        prev = temp;
        temp->next = scan;
    }
    else  
    {  
        prev = scan;
        scan = scan->next;
    }

    if (l2 != NULL)  
    {  
        prev->next = l2;
    }

    return (top);
}

The same problem applied to the case of arrays can be encoded as follows:

void MergeSort(int array[], int size)  
{  
    int size1,size2;
    int *arr1, *arr2;

    if (size <= 1)  
    {return;
    
    size1 = size / 2;
    size2 = size - size1;

    arr1 = CopyArray(array,0,size1);
    arr2 = CopyArray(array, size1, size2);
    MergeSort(arr1,size1);
    MergeSort(arr2, size2);
    Merge(arr1, arr2, top);"
void MergeSort(int arr2[], size2);
void Merge(int array[], arr1, size1, arr2, size2);
free(arr1);
free(arr2);
}

void Merge(int destination[], int source1[], int size1,
            int source2[], int size2)
{
    int p, p1, p2;
    p = p1 = p2 = 0;
    while (p1 < size1 && p2 < size2)
    {
        if (source1[p1] < source2[p2])
            {
                destination[p] = source1[p1];
                p++;
                p1++;
            }
        else
            {
                destination[p] = source2[p2];
                p++;
                p2++;
            }
    }
    while (p1 < size1)
    {
        destination[p] = source1[p1];
        p++;
        p1++;
    }
    while (p2 < size2)
    {
        destination[p] = source2[p2];
        p++;
        p2++;
    }
}

int *CopyArray(int source[], int from, int size)
{
    int i;
    int *new;

    new = NewArray(to, int);
    for (i = 0; i < to; i++)
        new[i] = source[from + i];
    return new;
}

The complexity of the merge sort process can be measured to be in the order of \(O(n \log n)\).

4 Radix Sort

Radix sort is based on comparing individual digits of the numbers in the sequence.
Consider the sequence

362, 745, 885, 957, 054, 786, 080, 543, 012, 565

We treat the numbers as sequences of characters and we split them in 10 sublists depending on the last digit:

[0] -> 080
[1]
[2] -> 362, 012
[3] -> 543
[4] -> 054
[6] -> 786
[7] -> 957
[8]
[9]

And then the sublists are put together again in the same order

080, 362, 012, 543, 054, 745, 885, 565, 786, 957

The same process is repeated but this time considering the second digit from the right:

[0]
[1] -> 012
[2]
[3]
[5] -> 054, 957
[7]
[8] -> 080, 885, 786
[9]

and again we put them back together:

012, 543, 745, 054, 957, 362, 565, 080, 885, 786

We repeat the process one more time using the leftmost digit:

[0] -> 012, 054, 080
[1]
[2]
[4]
[6]
[7] -> 745, 786
[8] -> 885
[9] -> 957

and putting them together we obtain:

012, 054, 080, 362, 543, 565, 745, 786, 885, 957