CHAPTER 2: SORTING ALGORITHMS

Contents

1. Presentation

Sorting is one of the most fundamental problems in algorithms. After sorting many other problems become easy to solve such as unicity and search.

Sorting consists of rearranging a list of n objects in such a way:

$$X_1 \leq X_2 \leq \cdots \leq X_n \text{ : Ascending order sort}$$
$$X_1 \geq X_2 \geq \cdots \geq X_n \text{ : Descending order sort}$$

There is numerous sorting methods, such as:

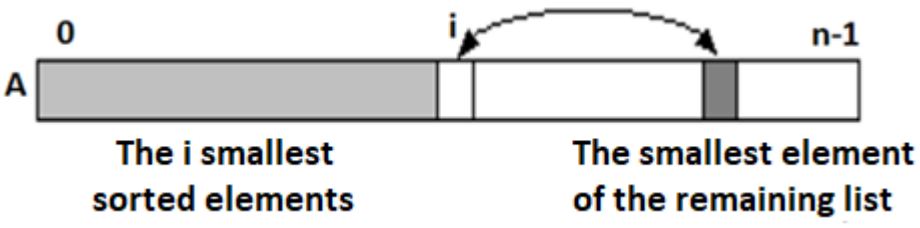
- Selection sort
- Insertion sort
- Bubble sort
- Merge sort
- Quick sort
- Shell sort
- …

In what follows, we describe the main sorting algorithms and analyze their time complexity.

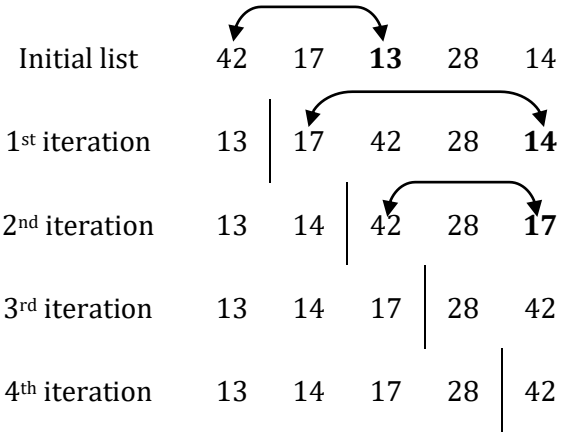
2. Selection Sort

2.1. Principle

Iteratively, sorting by selection consists of finding the smallest element and then putting it at the beginning.



Example



2.2. Implementation

```
void selectionSort(int *A, int n)
{
for (int i=0 ; i<n-1 ; i++)
 { int imin=i;
   for (int j=i+1 ; j<n ; j++)
       if (A[j]<A[imin]) imin=j;
   swap(A,i,imin);
 }
}
```

2.3. Complexity analysis

The worst case and the best case are the same, since to find the smallest element, $(n-1)$ iterations are necessary, for the 2nd smallest element, $(n-2)$ iterations are performed… until the penultimate smallest element that requires 1 iteration. The total number of iterations is therefore:

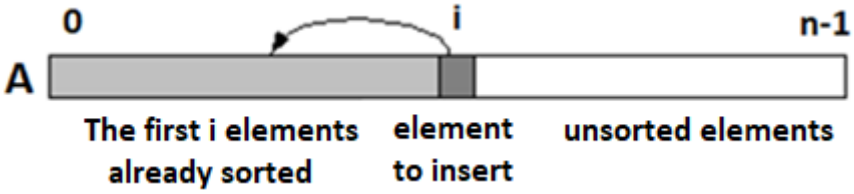
$$\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = O(n^2)$$

It remains that selection sort perform only $O(n)$ permutations.

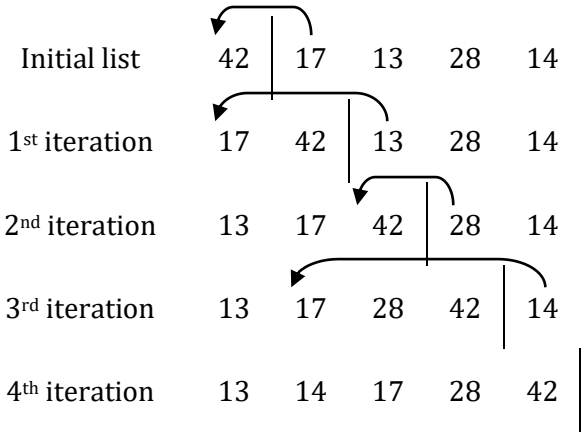
3. Insertion sort

3.1. Principle

Iteratively, we insert the next element in the part that has already been sorted. The starting part that is sorted is simply the first element.



Example



3.2. Implementation

```
void insertionSort(int *A, int n)
{
 for (int i=1 ; i<n ; i++)
 { int temp=A[i], j=i;
   while (j>0 && A[j-1]>temp)
   { A[j] = A[j-1]; j--;}
   A[j] = temp;
 }
}
```

3.3. Complexity analysis

Since we don't necessarily have to scan the whole part already sorted, the worst case and the best case are different.

Best case: if the array is already sorted, each element is always inserted at the end of the sorted part; we don't have to move anything. As we have to insert $(n-1)$ elements, each generating only one comparison, the complexity is thus $O(n)$.

Worst case: if the array is inversely sorted, each element is inserted at the beginning of the sorted part. In this case, all the elements of the sorted part must be moved at each iteration. The ith iteration generates $(i-1)$ comparaisons and permutations:

$$\sum_{i=1}^{n}(i-1) = \frac{n(n-1)}{2} = O(n^2)$$

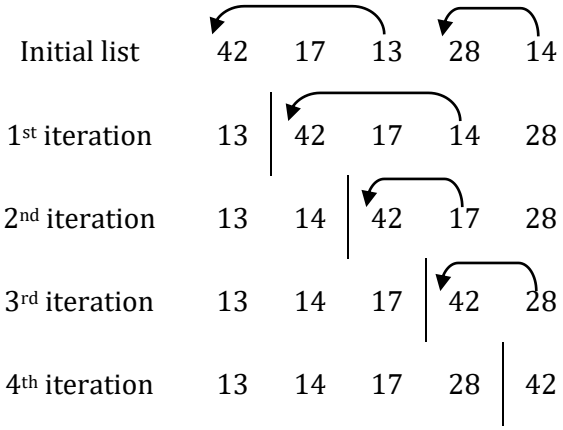
Note for this algorithm that the number of permutations performed in the worst case is of the order of $O(n^2)$.

4. Bubble Sort

4.1. Principle

Go through the table by comparing the successive elements two by two and swap if they are not in order.

Example



4.2. Implementation

```
void bubbleSort(int *A, int n) {
for (int i=0 ; i<n-1 ; i++)
  for (int j=n-1 ; j>i ; j--)
    if (A[j]<A[j-1]) swap(A,j,j-1);
}
```

4.3. Complexity analysis

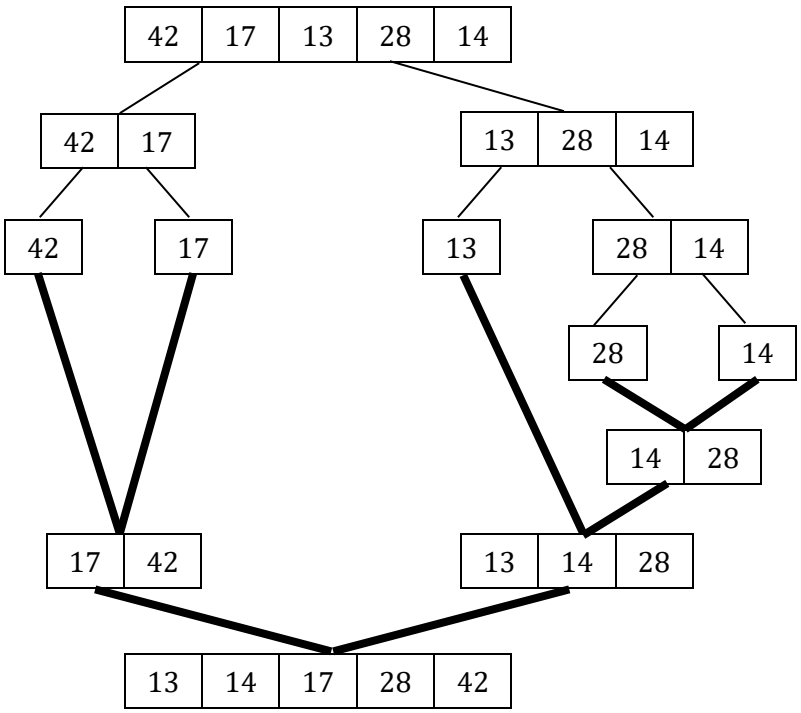
Overall, it is the same complexity as selection sort. The best and the worst case are the same with a complexity of $O(n^2)$. Except that compared to selection sort, the amount of permutations performed in this algorithm is of the order of $O(n^2)$.

5. Merge Sort

5.1. Principle

This algorithm divides the list into two parts. After these 2 parts are sorted (usually recursively), they are finally merged to obtain a sorted list. This illustrates well the famous strategy "Divide and Conquer".

Example



5.2. Implementation

```
void mergeSort(int *A, int istart, int iend)
{
 int imedian;
 if (istart<iend) {
   imedian=(istart+iend)/2;        (* DIVIDE  *) ..... O(1)
   mergeSort(A,istart,imedian);    (* CONQUER *) ..... T(n/2)
   mergeSort(A,imedian+1,iend);    (* CONQUER *) ..... T(n/2)
   merge(A,istart,imedian,iend);   (* COMBINE *) ..... O(n)
  }
}

void merge(int *A, int istart, int imedian, int iend)
{
  int n1,n2,i,j;
  int* R = new int[50]; int* L = new int[50];
  n1=imedian-istart+1;
  n2=iend-imedian;
  for (i=1;i<=n1;i++) L[i]=A[istart+i-1];
  for (j=1;j<=n2;j++) R[j]=A[imedian+j];
  L[n1+1]=9999;  //9999 = very large number
  R[n2+1]=9999;
  i=j=1;
  for (int k=istart;k<=iend;k++) {
  if (L[i]<=R[j]) {A[k]=L[i];i++;}
  else {A[k]=R[j];j++;}
 }
}
```

5.3. Complexity analysis

Merge sort complexity can be expressed by the following recurrence equation:

$$\mathrm{T(n)} = \begin{cases} O(1) & if\ n = 1 \\ 2T(n/2) + O(n) & if\ n > 1 \end{cases} \quad \Rightarrow \quad \mathrm{T(n)} = \mathrm{O(n log_2 n)}$$

Or, we can say that the array A will be divided by 2 until obtaining arrays of size 1, thus:

Size of A: $n, n/2, n/4, \ldots, n/2^p$ with $n/2^p = 1 \Rightarrow p = log_2(n)$

Since at each step, one (or more) merge operation(s) which takes $O(n)$ operations is performed on sub-arrays, we obtain $O(nlog_2 n)$ complexity.

6. Quick sort

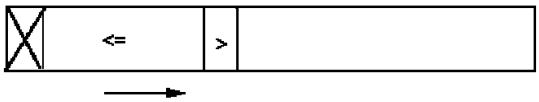
6.1. Principle

The key idea of this algorithm is to divide the array into two parts separated by an element called **pivot** in such a way that the elements of the left part are all less than or equal to this element and those of the right part are all greater than the pivot. This fundamental step in quicksort is called **partitioning**.

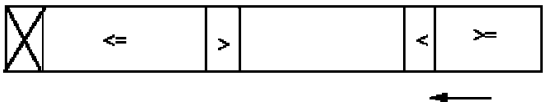
Pivot selection: The ideal choice would be when the table will be divided into two equal parts (see complexity analysis), but this is not always possible. We can take the first or the last or even select randomly the pivot.

Partitioning:

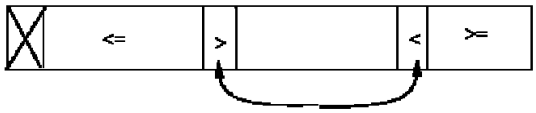
- We go from **left to right** until we find an element **greater** than the pivot.



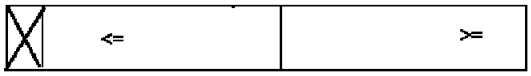
- We go from **right to left** until we find an element **lower** than the pivot.



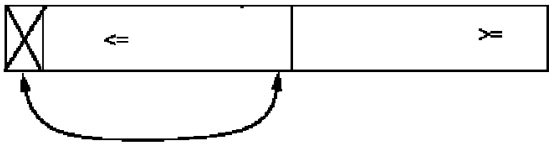
- We perform a swapping between these two elements.



- Repeat the left-right and right-left exploring until we obtain:

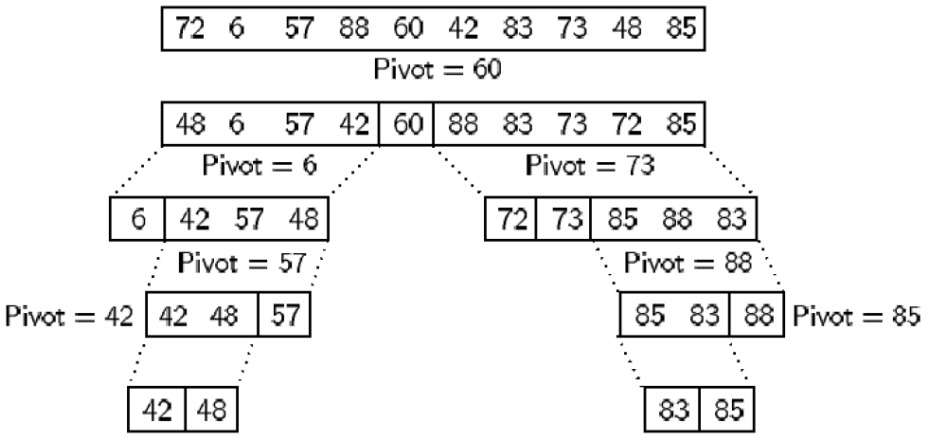


- Finally, we put the pivot at the border by swapping.



In the following (example & implementation) we choose the element in the middle of the array as the pivot.

Example



6.2. Implementation

```
void quickSort(int *A, int istart, int iend)
{
 int ipivot;
 if (istart<iend) {
   ipivot=partition(A,istart,iend);
   quickSort(A,istart,ipivot-1);
   quickSort(A,ipivot+1,iend);
 }
}

int partition(int *A, int istart, int iend)
{
 int pivot,aux,i,j;
 pivot=A[(istart+iend)/2];
 i=istart;j=iend;
 while (i<j)  {
   while (pivot>A[i]) i++;
   while (pivot<A[j]) --j;
   if (i<j) swap(A,i,j);
 }
 return j;
}
```

6.3. Complexity analysis

- **Favorable case:**

The best thing that can happen is that every time the Partition() function is called, it exactly divides the array (or sub-array) into 2 parts of the same size.
On the first pass, the n elements of the array are compared with the pivot value to swing them to the right or to the left. So there are n comparisons. On the second pass there are 2 Partition() functions which each perform their role on their half of the array. Each function must compare the $n/2$ elements of the subarray to perform the swing. So in fact, there are still n comparisons for this pass. It will be the same for the other iterations.
We can express the complexity by the following formula:

$$T(n) = \begin{cases} O(1)\ if\ n = 1 \\ 2T(n/2) + O(n)\ if\ n > 1 \end{cases} \quad \Rightarrow \quad T(n) = O(nlog_2 n)$$

- **Unfavorable case:**

The worst thing that can happen is that each time the Partition() function is called, it places the entire sub-array to the right or to the left (of course excluding the pivot element which is then in its final place). In this unfavorable case, the quick sort turns into a bubble sort.
On the first pass, there will therefore be n comparisons. At the second pass there is already an ordered value and a sub-array of $n-1$ elements, so there will be $n-1$ comparisons performed by the Partition() function. At the third pass, there will be $n-2$ comparisons and so on.
To perform a complete sort, the total amount of comparisons is then:

$$n + (n-1) + \ldots + 2 + 1 = n(n+1)/2$$

Which gives a O(n^2) worst case complexity for the Quicksort method. But it remains that the average case complexity is O($n\log n$).