SHORT COMMUNICATION

INDEX SORT ALGORITHM FOR POSITIVE INTEGERS

ADEWUMI, S. E

Department of Mathematics
University of Jos, Nigeria
adewumis@gmail.com

Sorting algorithms have been described variously by (Hubbard, 2000; Lipchutz, 2002). Sorting is required in database application to arrange items in either ascending or descending order of magnitude. It is sometimes required to determine the magnitude of a number say the ages of persons, census data, budgetary figures etc.

In all these cases, sorting may be used to achieve the arrangement in order of magnitude. The applications of sorting algorithm are numerous and cannot be exhausted in this paper. An efficient algorithm will provide better sorting time and easy of implementation. We described some popular sorting algorithm of interest, the purpose of which is to provide bases for comparison with our method.

Bubble sort

Suppose we have a list of numbers denoted by $a_1, a_2, \ldots, a_n$ an in memory. Bubble sort algorithm proceeds through a sequence of iterations, each time moving the next largest item into its correct position. During each iteration, pair of consecutive elements are compared in order to move the larger element up. The complexity of bubble sort is naturally determined by the number of comparison experienced during each pass, (Lipschtz, 2002). Therefore in the first pass we have n-1 comparisons, second pass n-2 comparisons and son on. Thus the complexity of bubble sort is

$$f(n) = (n-1) + (n-2)\cdots + 2 + 1 = \frac{n(n-1)}{2}$$

$$= \frac{n^2}{2} + O(n) = O(n^2)$$

Insertion sort

Suppose we have a list of numbers denoted by $a_1, a_2, \ldots, a_n$ in memory. Insertion sort algorithm proceeds through a sequence of scans on the array A from $a_1, a_2, \ldots, a_n$, inserting each element a_k into its proper position. It takes an initial unsorted sequence $S = \{s_1, s_2, \cdots, s_n\}$ and computes a series of sorted sequences $S'_0, S'_1, \cdots, S'_n$ (Preiss, 2000). The complexity for this algorithm is also

$$f(n) = (n-1) + \cdots + 2 + 1 = \frac{n(n-1)}{2}$$

$$= \frac{n^2}{2} + O(n) = O(n^2)$$

Selection Sort

Suppose we have an array A denoted by elements $a_1, a_2, \ldots, a_n$ in memory. Selection sort algorithm works by first finding the smallest element in the list and put it in the first position. Then find in the second element in the list and put it in the second position and so on. The complexity of its runningtime is the same as insertion sort described earlier.

Merge sort

Suppose an array A with n elements $a_1, a_2, \ldots, a_n$ in memory. Merge sort algorithm works by the use of divide-and-conquer algorithm. It proceeds by taking pair of elements, sort them and merge it with another sorted pair and sorting this pair. After some pass K, the array will be partitioned into sorted subarrays where each subarray, except possibly the last, will contain exactly 2^k elements. Therefore the algorithm will require at most (worse-case) log *n* passes to sort an *n*-element array A.

THE APPROACH

Suppose we have some elements denoted by $a_1, a_2, \ldots, a_n$ in memory. This new algorithm works by storing these elements in another array say A, with each integer stored as their index. With this algorithm, an integer number say 10 will be stored in position 10 of the array, a number say 3 will be in position 3 of the array and so on. This will proceed until all the numbers have been fixed in their rightful position in the array. That is, numbers are merely copied into their index position. Once this has been completed, the numbers are then displayed the way they appear in the array; as they have been naturally sorted in their order of magnitude.

If for example, we have numbers 11, 3, 5, 2, 1, 8 in memory to be sorted. This algorithm works by declaring an array say A and then assign each element to their respective index as shown in Table 1.

TABLE 1. ARRAY A WITH THE ELEMENTS IN THEIR RESPECTIVE INDEX

A(0)	
A(1)	1
A(2)	2
A(3)	3
A(4)	
A(5)	5
A(6)	
A(7)	
A(8)	8
A(10)	
A(11)	11

After this storing, we copy out by printing the content and it can be readily observe that every integer number has been sorted naturally in the way they appear. If the content of the array above is printed, we have the following 1, 2, 3, 5, 8, 11 as the result of sorting the data in the example. This is a fast means of sorting integer numbers. The only overhead cost we envisaged is in term of memory required for the array when the numbers to be sorted is large. We are also aware that memory module gets cheaper by the day, so this may not be a major challenge as such. The index sort algorithm is further described below.

The Index Sort Algorithm

Elements in memory are stored as the index for the position of each number. Assuming the elements in memory are in array a[*n*], we store this elements into another array b[n] by assigning each element a[*i*] to variable say *k*, that is, *k*=a[*i*], *i* starting from 0, 1, …, n .

Write each element into their respective position in array B, that is b[*k*] = *k*.

Traverse array B and copy out the data leaving out indexes where the content is zero.

Specifically, the following program segment will sort any unsorted positive integer numbers in array A into array B:

```
for (j=0;j<=m;++j)
        {
      k=a[j];
     b[k]=k;
        }
```

A complete c++ program has been developed and tested based on this algorithm and it has performed successfully well.

The c++ pseudo-code for achieving this will be:

```
#include <iostream>
void indexsort(int [])
int main()
{
int x, int a[];
cout <<"Enter a positive integer as the length of data to be sorted: ";
cin >> x;
cout << "Enter " << x << "numbers you wish to sort" << endl;
for (int i=1; i<=x; i++)
{
int n;
cout << "Now enter the numbers one after the other: " cin>> n;
if (n==0)
a[n-1] = 1;
else
a[n-1] = n;
}
for (int i=1; i<=x; i++)
{
if (a[i-1] = 1)
cout << 1;
else if (a[i-1] > 0)
cout << a[i-1];
}
}
```

Algorithm analysis for this method

The main segment of the program that does the storing and copying is shown below while the program analysis follows:

```
1. for (j=0;j<=m;++j)
2.  {
3.          k=a[j];
4.          b[k]=k;
5   }
```

If we take each statement line by line, we obtain the following in Table 2.

TABLE 2. ALGORITHM ANALYSIS OF THE SCHEME

Statement	Detailed running time	Simple analysis
1a	$t_{fetch} + t_{store}$	2
1b	$(2t_{fetch}+t_{<}) \times (n+1)$	3(n+1)
1c	$(2t_{fetch} + t_{+} + t_{store}) \times n$	4n
3	$3t_{fetch} + t_{[.]} + t_{store}$	5
4	$t_{fetch} + t_{store}$	2
	$(4t_{fetch} + t_{+} + t_{<} + t_{store}) \times n$ $+7t_{fetch} + t_{+} + t_{<} + 3t_{store}$	7n + 12

From the simple program analysis above, we have *7n+12* which shows that the running time complexity is of order *O(n)*, that is, it is linear.

CONCLUSION

We have been able to devise a new sort algorithm and have been able show that the complexity of this algorithm is *O(n)*. This method will provide the efficiency required for sorting data in linear time.

REFERENCES

Hubbard J. R. (2000). *Schaum's Outlines of Programming with C++*. McGraw-Hill Publishing Company

Lischutz, S. (2002). *Schaum's Outlines of Theory and Problems of Data Structures*. Tata McGraw-Hill Publishing Company Limited.

Preiss B.R. (2000). *Data Structure and Algorithm With Object-Oriented Design Patterns in C++.* John Wiley & Sons Inc.,