International Journal of Computer Applications (0975 - 8887)
Volume 143 - No.9, June 2016

Matrix Sort - A Parallelizable Sorting Algorithm

S. Kavitha
Professor,
Computer Science and Engineering,
SSN College of Engineering,
Rajiv Gandhi Salai, Chennai-603110

Vijay V.
Computer Science and Engineering,
SSN College of Engineering,
Rajiv Gandhi Salai, Chennai-603110

Saketh A. B.
Computer Science and Engineering,
SSN College of Engineering,
Rajiv Gandhi Salai, Chennai-603110

ABSTRACT

Sorting algorithms are the class of algorithms that result in the ordered arrangement of a list of given elements. The arrangement can be in ascending or descending order based on the requirement given. Time complexity, space complexity and optimality are used to assess the algorithms. In this paper, a new sorting algorithm called Matrix sort is introduced. This algorithm aims to sort the elements of a matrix without disturbing the matrix structure. It has a time complexity of O(n $\sqrt{n}\log\sqrt{n}$) and hence takes lesser time than existing O(n^2) algorithms. A pseudocode for the algorithm is provided and the best, average and worst case time complexities are derived.

General Terms

Algorithm, Sorting,Time Complexity, Space complexity, Comparison

Keywords

Matrix sort, Top-down, Parallelizable sorting

1. INTRODUCTION

Sorting algorithms are widely used in a variety of computer applications. These applications span a wide range of fields like operations research, business solutions, image processing, data mining, numerical methods, government applications and so on. Matrix sort algorithm can be used where the elements of a matrix need to be searched without disturbing its structure.

The two classes of sorting algorithms are internal sorting algorithms and external sorting algorithms. An internal sorting algorithm is one which can take place entirely within the main memory of a system. Some common internal sort algorithms are Bubble sort, Quick sort and Selection sort. External sorting algorithms are those algorithms which can work on data that does not fit entirely in the main memory. Part of the data might reside in a slower external memory. A famous external sorting algorithm is the merge sort algorithm.

Table-1 represents the various sorting algorithms, and their time complexities[3]. It comprises the best case , average case and worst case time complexities. It also specifies the memory requirements.

Table-1 Comparison of sorting algorithms

Name	Best Case	Average Case	Worst case	Memory	Stable
Bubble sort	n	n^2	n^2	1	Yes
Insertion sort	n	n^2	n^2	1	Yes
Selection sort	n^2	n^2	n^2	1	No
Quicksort	n logn	n logn	n^2	logn	in-place version is unstable

The core operation of the matrix sort algorithm is called the top-down operation. It compares every element in the second half of a row with the elements in the first half of it's previous row. This is the novel operation that is proposed in an attempt to reduce the time complexity.

2. RELATED WORK

Sorting algorithms have been around since the 1940s. Algorithms which result in a permutation of the given input in which the input is arranged in the desired order are called sorting algorithms. Several approaches have been proposed for sorting the given input. A few of the most prominent sorting algorithms used widely are discussed in this section.

2.1 Bubble Sort

Bubble sort is a sorting algorithm that works by bubbling the largest elements to the end of the list.There are several approaches to optimize the algorithm by stopping once the list has been sorted. For example, the algorithm can be stopped if no swaps were performed in an iteration, but that should be implemented in a different manner. The most common and widely used implementation of bubble sort has been presented below. The pseudocode for bubble sort is as follows:

```
procedure BubbleSort:
var j,t: integer;
   begin
   repeat
      t:=a[1];
         for j:=2 to N do
            if a[j-1] > a[j] then
               t:=a[j-1];        a[j-1]=a[j];        a[j]=t;
   until t=a[1];
   end;
```

2.2 Insertion Sort

Insertion sort is a simple sorting algorithm, which keeps inserting elements in the right place. The algorithm proceeds by considering one element at a time, inserting it in its proper place among those that have already been considered(sorted). The pseudocode for insertion sort is as follows:

```
procedure InsertionSort;
var i,j,v:integer;
   begin
   for i:=2 to N do
      begin
      v:=a[i]; j:=i;
      while a[j-1]>v do
         a[j]=a[j-1]; j:=j-1;
         end;
      a[j]:=v;
      end;
   end;
```

While considering an item, all the items larger than the one under consideration are moved one position to the right. This item is then placed in the vacant position.

2.3 Selection sort

Selection sort works by finding the i^{th} smallest element in the i^{th} iteration, and places it in the correct position. Initially, it finds the smallest element and swaps it with the element in the first position. It then finds the second smallest element and swaps it with the element in the second position and so on. The pseudocode for selection sort is as follows:

```
procedure SelectionSort;
var i,j,min,t:integer;
   begin
   for i:=1 to N do
      begin
      min:=a[i]
      for j:=i+1 to N do
         if a[j]<min then min:=j;
         end;
      t:=a[i]; a[i]:=a[j]; a[j]:=t;
      end;
   end;
```

2.4 Quicksort

Quicksort is a sorting algorithm that is based on the divide and conquer strategy. Quicksort works by dividing an array into subarrays, and sorting these subarrays respectively. Quicksort can be implemented using recursion. It partitions the array into two partitions, one which has elements larger than the chosen pivot, and the other which has elements smaller than the pivot. Then quicksort is called for these two partitions. Quicksort is one of the most widely used algorithms. The pseudocode for quicksort is as follows:

```
procedure QuickSort(l,r:integer);
var i:integer;
   begin
   if l<r then
   begin
      pivot:=a[r] i:=l-1; j:=r;
         repeat
            repeat i:=i+1 until a[i]<pivot;
            repeat j:=j-1 until a[j]>pivot;
            t:=a[i]; a[i]=a[j]; a[j]=t;
         until j<=i;
         a[j]:=a[i]; a[i]:=a[r]; a[r]:=t;
         QuickSort(l,i-1)
         QuickSort(i+1,r)
      end;
   end;
```

There are several ways to implement quicksort. The most common and naive approach to implement quick sort has been discussed above.

2.5 Merge sort

Merge sort is an external sorting algorithm invented by Jon Von Neumann. Merge sort algorithm is based on the divide and conquer strategy. The idea behind merge sort is to split a list of elements to be sorted into two smaller lists of the same size, and then apply merge sort recursively on these smaller lists.This split continues until the list contains zero or one elements. The merge operation can be performed by stepping through the lists in linear time. Merge sort always runs in O(n logn) time, but requires O(n) space. Merge sort is implemented using two procedures, one which splits the list into two halves, and the other that merges the two lists and hence sorts them. In the given pseudocode, the procedure mergesort(A:array) splits the list(A) and the procedure merge(A,B:array) merges the two lists(A,B). The pseudocode for merge sort is as follows:

```
procedure mergesort(A:array)
var l1,l2:array;
   begin
   if n==1 return A
   l1=A[0]..A[n/2]
   l2=A[n/2 + 1]..A[n]
   l1=mergesort(l1)
   l2=mergesort(l2)
   return merge(l1,l2)
   end;
```

```
procedure merge(A,B:integer);
var C:array;
var i,j,k:integer;
```

```
begin
i:=0; j:=0; k:=0;
while((i<n_A) and (j<n_B))
   if(A[i] < B[j])
      C[k++]:=A[i++]
   else
      C[k++]:=B[j++]
while(i<n_A)
   C[k++]:=A[i++]
while(j<n_B)
   C[k++]:=B[j++]
return C
end;
```

3. MATRIX SORT ALGORITHM

3.1 Working

The working of the matrix sort algorithm is described in this section. The algorithm takes as input a 2D array of elements and sorts it in the desired order. The elements are present in the desired order once the algorithm stops.

Implementation strategy:

i. Rows of the matrix are first sorted.
ii. Then the columns of the matrix are sorted.
iii. The following "Top-down" operation is then performed 'n' times on all the 'n' rows of the matrix:

a) Every j^{th} element in the first half of the current(i^{th}) row is compared with the $(n-j)^{th}$ element in the previous row.

b) If their order is opposite to the desired order, then the elements are swapped.

c) Rows of the matrix are sorted.

The top-down operation is performed since the largest element in a row can be smaller than the smallest element in it's next row. Since the elements of the first half of any row are smaller than those in the first half of its next row (as a result of the columns having been sorted), it is sufficient for the elements of the second half of the i^{th} row to be compared with the elements of the first half of the $(i+1)^{th}$ row. Since an element that should be present in the first row can be present in the last row initially, top-down operation has to be performed 'n' times.

3.2 Pseudocode

This section contains the pseudocode for the matrix sort algorithm.

```
procedure MatrixSort(N:integer,A:integer[N][N])
Input: A matrix with elements to be sorted
Output: Elements in sorted order
Algorithm to sort elements in the matrix
   var i,j,flag;
   begin
      for i:=1 to N do
         begin
         RowSort(A,i); //sorts the elements in the i^th row
                          using quicksort
         end
      for j:=1 to N do
         begin
         ColumnSort(A,j); //sorts the elements in the j^th column
                             using quicksort
         end
      for k:=1 to N do
         begin
         flag=TopDown(A,N); //call to procedure that performs
                               top-down operation
         if flag==1 then break;
         else SortAllRows(A); //call to sort all rows in the matrix
                                 using quicksort
         end
   end
```

The pseudocode for the top-down operation is given below.

```
procedure TopDown(A,N)
Input: Matrix to perform top-down operation
Output: Matrix after top-down operation
Algorithm to perform top-down operation
      var i,j,flag;
      begin
      flag=0;
      for i:=1 to N do
         for j:=1 to N/2 do
         begin
            if A[i][j]<A[i][n-j-i] then
               swap A[i][j],A[i][n-j-i];
               flag=1;
         end
      if flag==0 then
         return 1;
      else
         return 0;
      end
```

The above mentioned N, in the algorithm is the square root of the total number of elements in the input. The input is stored in an N x N matrix, and hence contains N^2 elements. While sorting the rows, each row can be sorted using a separate processor. The same applies for columns too. While sorting the columns, all columns can be sorted independent of the other columns parallely.

Example for illustrating the working of matrix sort

$$\begin{pmatrix}7&2&5\\8&4&3\\6&1&9\end{pmatrix}$$ (1.a)

$$\begin{pmatrix}2&5&7\\3&4&8\\1&6&9\end{pmatrix}$$ (1.b)

$$\begin{pmatrix}1&4&7\\2&5&8\\3&6&9\end{pmatrix}$$ (1.c)

$$\begin{pmatrix}1&4&2\\7&5&8\\3&6&9\end{pmatrix}$$ (1.d)

$$\begin{pmatrix}1&4&2\\7&5&3\\8&6&9\end{pmatrix}$$ (1.e)

$$\begin{pmatrix}1&2&4\\3&5&7\\6&8&9\end{pmatrix}$$ (1.f)

$$\begin{pmatrix}1&2&3\\4&5&6\\7&8&9\end{pmatrix}$$ (1.g)

$$\begin{pmatrix}1&2&3\\4&5&6\\7&8&9\end{pmatrix}$$ (1.h)

$$\begin{pmatrix}1&2&3\\4&5&6\\7&8&9\end{pmatrix}$$ (1.i)

Fig.1 Example for illustrating the working of matrix sort

The unordered input matrix is shown in Fig.(1.a). Let's look at the various steps involved below.

(i) Each row of the matrix(A) is sorted. The matrix whose rows are

sorted is shown in Fig.(1.b)
(ii) Each column of the matrix is sorted. The matrix whose columns are sorted is shown in Fig.(1.c)
(iii) The top-down operation is performed. Since A[2][1] < A[1][3], these elements are swapped.
A[2][2] > A[1][2], and hence these elements are not disturbed. The matrix after top-down for the second row is shown in Fig.(1.d)
(iv) In similar manner, top-down operation is performed for the third row. The matrix after top-down operation for the thid row is shown in Fig.(1.e)
(v) The top-down operation has completed one iteration, and hence the rows have to be sorted. The matrix whose rows are sorted after top-down operation is shown in Fig.(1.f)
(vi) Since swaps were performed in the previous top-down, the algorithm does not stop. The matrix after the next top-down iteration is shown in Fig.(1.g)
(vii) The matrix whose rows are sorted after the second top-down is shown in Fig.(1.h)

Since no swaps are performed now, the result has been obtained. The resultant matrix which is completely sorted(in the desired increasing order) is shown in Fig.(1.i)

Hence, the matrix has been sorted by the matrix-sort algorithm. If the sorted matrix has to be stored in a 1-D array, the elements can be unrolled one by one from the first row to the last row into a 1-D array. This list is a sorted permutation of the given input.

4. ANALYSIS OF MATRIX SORT

In this section, a detailed analysis of the time complexity of matrix sort is presented. For this analysis, a naive sequential implementation is considered, since the performance on a parallel processor will depend on the processor's specification.

Consider the input size to be n elements. For convenience, it is assumed that n is a perfect square, failing which the matrix can be padded with multiple copies of an extra element to make it a perfect square. Since the total number of elements is n, the size of the matrix is $\sqrt{n}$ x $\sqrt{n}$.

It is well known that the average case time complexity of Quicksort is n$\log n$. Each row in the input matrix consists of $\sqrt{n}$ elements. Hence, the rows can be sorted in $\sqrt{n}\ \log\sqrt{n}$ iterations. Each of the columns also take $\sqrt{n}\ \log\sqrt{n}$ iterations.
Hence, for row and column sorts,

$$T_1(n) = \sqrt{n} * (\sqrt{n}\log\sqrt{n}) + \sqrt{n} * (\sqrt{n}\log\sqrt{n}) = 2n\log\sqrt{n} \quad \ldots (1)$$

The above expression is the time taken for sorting the rows and columns initially. Once this is done the top-down operations are performed.

While performing the top-down operation for the i^{th} row, the first half of this row is compared with the second half of the $(i-1)^{th}$ row (previous row). Hence, $\sqrt{n}/2$ comparisons (half the number of elements in the row) are performed. Since this is repeated $\sqrt{n}$ times, the number of comparisons is, $\sqrt{n} * \sqrt{n}/2 = n/2$ comparisons. This operation has to be performed for $\sqrt{n}$ rows. Hence, the number of comparisons becomes $n\sqrt{n}/2$.

This results in the following expression,

$$T_2(n) = 2n\log\sqrt{n} + n\sqrt{n}/2 \quad \ldots (2)$$

Each call to top-down is followed by a sort of all rows. Hence, this adds n $\log\sqrt{n}$ for each call to top-down. Hence, for $\sqrt{n}$ calls, the time complexity is $n\sqrt{n}\log\sqrt{n}$.
Hence, the final expression for the time complexity is as follows:

$$T(n) = 2n\log\sqrt{n} + n\sqrt{n}/2 + n\sqrt{n}\log\sqrt{n} \quad \ldots (3)$$

Hence, Eqn.(3) is the expression for the time complexity of matrix sort. In Big-oh notation, the time complexity for the matrix sort algorithm is,

$$T(n) = O(n\sqrt{n}\log\sqrt{n}) \quad \ldots (4)$$

Hence, this algorithm is proven to be better than other sorting algorithms that have a quadratic time complexity, i.e, $O(n^2)$. Thus, an expression for the time complexity of matrix sort is arrived at.

5. RESULTS

For any sorting algorithm, the results convey how useful it is for real world applications. In this section the every case analysis of matrix sort has been discussed. A comparison of the running time of matrix sort with other sorting algorithms has also been presented.

5.1 Every case analysis

This section presents the every case analysis of matrix sort, i.e, the best case, average case and worst case time complexity[10] for the matrix sort algorithm. Each of these relations are derived one by one.

Best case

The best case input is a matrix of elements which is sorted. Consider an input of n elements. For this input, all the rows of the matrix are sorted, then all the columns are sorted. Since there are no disorders, there are no swap operations and hence the algorithm concludes that the matrix has been sorted, and stops.

For sorting $\sqrt{n}$ rows, the time complexity is $\sqrt{n} * (\sqrt{n}\log\sqrt{n})$. Sorting $\sqrt{n}$ columns also has the same time complexity. The top-down operation performs n/2 comparisons, and then the algorithm stops. Hence, the best case time complexity is,

$$T_{best}(n) = \sqrt{n} * (\sqrt{n}\log\sqrt{n}) + \sqrt{n} * (\sqrt{n}\log\sqrt{n}) + n/2 \quad \ldots(5)$$
$$= 2n\log\sqrt{n} + n/2 \quad \ldots(6)$$
$$\text{Hence, } T_{best}(n) = O(n\log\sqrt{n}) \quad \ldots(7)$$

Average case

The average case input is the most common type of input. It can be any randomly ordered collection of elements stored in the matrix. The average case time complexity for matrix sort has already been derived in the analysis section of this paper. The average case complexity of the algorithm is,

$$T_{average}(n) = O(n\sqrt{n}\log\sqrt{n})$$

Worst case

The worst case input for this algorithm is when elements are sorted along the columns. Consider such an input matrix with n elements organized in $\sqrt{n}$ rows and $\sqrt{n}$ columns. For such an input, the top-down operation ends only in the $\sqrt{n}^{th}$ iteration, for an input matrix with n elements.

The worst case time complexity for quicksort is n^2, for n

elements. Hence, for one row or one column, the time complexity is n (since the number of elements in a row or a column is $\sqrt{n}$).

Hence, $T_3(n) = \sqrt{n} * n + \sqrt{n} * n = 2n\sqrt{n}$ (For $\sqrt{n}$ rows and $\sqrt{n}$ columns) ...(8)

The above expression is only for the row and column sorts. The top-down operation is performed $\sqrt{n}$ times. Each top-down operation performs n/2 comparisons, and a sort of all rows. Sorting all rows $\sqrt{n}$ times takes,

$T_4(n) = \sqrt{n} * \sqrt{n} * n$ (Since $\sqrt{n}$ rows are sorted $\sqrt{n}$ times, and the time complexity to sort a row with $\sqrt{n}$ elements is n) ...(9)

The final expression for the top-down operation alone is,

$T_5(n) = n^2 + n\sqrt{n}/2$ ... (10)

Hence, the worst case time complexity is,

$T_{worst}(n) = T_3(n) + T_5(n)$ ... (11)

The worst case time complexity of matrix sort is hence,

$T_{worst}(n) = 2n\sqrt{n} + n^2 + n\sqrt{n}/2$ ... (12)

Thus, $T_{worst}(n) = O(n^2)$ ... (13)

5.2 Comparison with existing algorithms

The following compares the running time of matrix sort algorithm with that of other sorting algorithms. The input was generated by a random number generator. The generated inputs have a size which is a perfect square. Zero padding can be done if the input is not a perfect square. The running time has been measured in microseconds.

Table-2 - Comparison of the running time of matrix sort with that of other sorting algorithms

Input size	Matrix sort	Bubble sort	Selection sort	Quicksort
100	153	105	72	29
400	818	622	708	140
900	1843	5731	3337	344
1600	3746	10719	8431	583
2500	8405	14552	14854	966
3600	20597	30677	20308	1477
4900	27058	45202	31317	1971
6400	45720	81933	46224	2735
8100	69753	127858	72779	1892
10000	102996	197263	104105	1968
10000	6120	155281	103515	144889

Fig.2 shows how the running time of various sorting algorithms varies with the input size. Fig.3 highlights the running time of the discussed sorting algorithms on a special case input, in which there is only one unique element.

From the obtained results, one can see that there are three classes of performance for matrix sort relative to other sorting algorithms. These three classes of performance are discussed here.

Class-1 From the first two rows of Table-2 i.e, for small sized inputs, the existing algorithms perform better than matrix sort. Also from Fig.2, it is evident that other algorithms perform better when the input is small sized. Hence, matrix sort is more

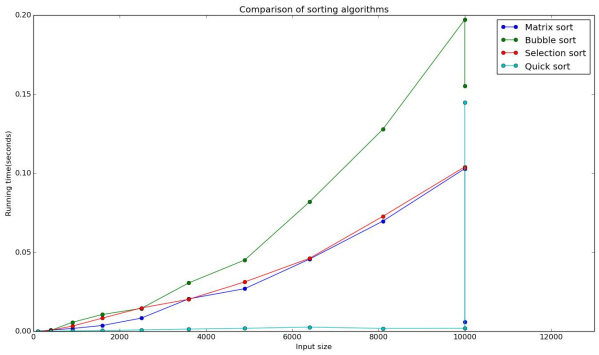

Fig.2 Comparison with existing algorithms

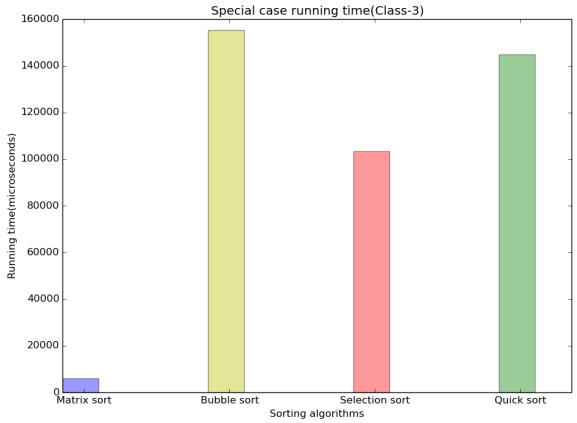

Fig.3 Special case running time(Class-3)

suitable for large-sized inputs. Sorting operations are generally performed on large sized inputs. Also, the time difference for small sized inputs is negligible, since it is of the order of microseconds.

Class-2 From the remaining rows and the middle region of Fig.2, it is clear that the matrix sort algorithm performs immensely better than other algorithms with a quadratic time complexity. Hence, extending this inference, a conclusion that matrix sort will perform incomparably better than algorithms of quadratic time complexity can be made, when the input is large sized. Since $\sqrt{n}$ is much larger than $\log\sqrt{n}$, **the matrix sort algorithm which has a time complexity of $O(n\sqrt{n}\log\sqrt{n})$ performs better than algorithms with a time complexity of $O(n^2)$**.

Class-3 (Special case) One of the worst case inputs for quicksort is when the input consists of only one unique element, for example, a list of ten 1's. Matrix sort algorithm beats Quicksort, which is considered to be one of the best sorting algorithms for this type of an input. This is evident from the last row of Table 2 for which the input was a matrix consisting of 10000 copies of the same element. In Fig.3, the running time of different sorting algorithms(including matrix sort) on this input has been presented. It is clear from Fig.3 that matrix sort takes the least time for running on this input compared to the other sorting algorithms considered. This shows that matrix sort will speed up sorting when there are very few unique elements in the given input.

6. CONCLUSION AND DISCUSSION

Hence, in this paper the working and results obtained for matrix sort algorithm have been discussed. Derivations for its best case, average case and worst case time complexities are presented. The matrix sort can be used in a plethora of applications spanning a multitude of fields. One major reason for it's application domain being huge is that this algorithm can be implemented parallely, which reduces the running time phenomenally. Since a wide range of applications demand parallel processing, this algorithm is highly suitable for a wide range of today's applciations, and those that are yet to be created. The simplicity of matrix sort algorithm is one of it's advantages. Several applications store data in two-dimensional data structures. Since the amount of data is too high, conversion of data structures is indeed a bottleneck for performance. Since the matrix sort algorithm uses a matrix as it's primary data structure, matrix sort is highly suitable for such applications. Thus, this is a very efficient algorithm that can dramatically reduce the time taken for one of the most important tasks accomplished by computers, sorting.

Sorting algorithms have applicability in many commercial and scientific applications in a wide variety of fields. Increasingly, most of these applications are being run on multi-processor systems and hence the parallelisation capabilities of algorithms have become important. Matrix sort uses a novel approach to sorting, where rows/columns are sorted independent of each other. Thus, there is great potential for parallelisation with a huge reduction in running times on multi-processor systems. The future scope of matrix sort includes the implementation of matrix sort in a completely parallel environment.

7. REFERENCES

[1] Donald Knuth. *The art of Computer Programming, Volume-3: Sorting and Searching, Third edition*. Addison-Wesley, Reading, Massachusetts, 1993.

[2] Hoare, C.A.R.(1961). *Algorithm 63,Partition; Algorithm 64,Quicksort;* Communications of the ACM, Vol.4, p.321

[3] Robert Sedgewick, Kevin Wayne. *Algorithms (4^{th} ed)*. Annalen der Physik, 322(10):891-921, 1905.

[4] Comparison of algorithms. `https://en.wikipedia.org/wiki/Sorting_algorithm`

[5] Anany Levitin. *Introduction to the Design & Analysis of Algorithms, 2nd Edition*.

[6] Mark Allen Weiss. *Data Structures and Algorithm Analysis in C++(Third edition)*. Addison-Welsey (2007)

[7] Robert Sedgewick(1978). *Implementing Quicksort programs*. Communications of the ACM 21

[8] Daniel H. Greene, Donald E. Knuth. *Mathematics for the analysis of algorithms*. Modern Birkhüser Classics (2007)

[9] Robert Sedgewick (1998). *Algorithms in C: Fundamentals, Data Structures, Sorting, Searching, Parts 1-4* (3^{rd} ed.). Pearson Education

[10] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C.Stein (2009)[1990]. *Introduction to Algorithms* (3^{rd} ed.). MIT Press and McGraw-Hill.

[11] N. G. de Bruijn(1958). *Asymptotic Methods in Analysis*. Amsterdam: North-Holland. pp. 5âĂŞ7.

[12] Papadimitriou, Christos H.(1994). *Computational complexity*. Reading, Mass.: Addison-Wesley.