**VISVESVARAYA TECHNOLOGICAL UNIVERSITY**

**Jnana Sangama, Belagavi-590018, Karnataka**



**Design and Analysis of Algorithms Assignment**

**on**

**“Hypercube: Odd-Even Merge Algorithm”**

**Submitted by**

**1BI19CS38 : Bharath Gowda B**

**For the academic year 2020-21**

****

**Department of Computer Science & Engineering**

**BANGALORE INSTITUTE OF TECHNOLOGY**

K.R. Road, V.V.Pura, Bengaluru-560 004

**INDEX**

* **INTRODUCTION Pg.02**
* **DESIgne technique Pg.02**
* **ALOGRITHM Pg.03**
* **IMPLEMENTATION Pg.04**
* **TIME ANALYSIS Pg.06**
* **APPLICATIONS Pg.08**
* **OUTPUT Pg.08**

**Introduction to the Algorithm**

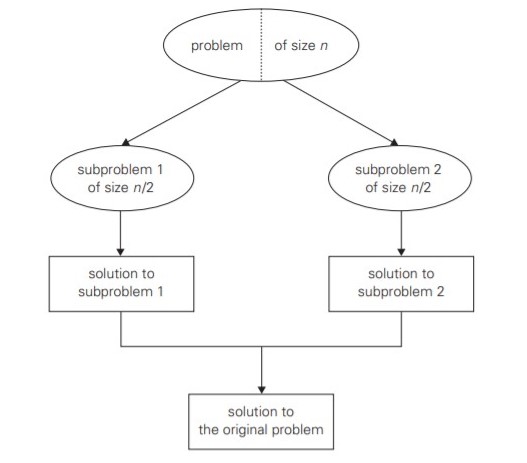
**Odd-Even Mergesort:**

The odd-even mergesort algorithm was developed by K.E. Batcher. It is based on a merge algorithm that merges two sorted halves of a sequence to a completely sorted sequence.

In contrast to [mergesort](https://www.inf.hs-flensburg.de/lang/algorithmen/sortieren/merge/mergen.htm), this algorithm is not data-dependent, i.e. the same comparisons are performed regardless of the actual data. Therefore, odd-even mergesort can be implemented as a [sorting network](https://www.inf.hs-flensburg.de/lang/algorithmen/sortieren/networks/sortieren.htm).

**DESIGNING TECHNIQUE**

Odd-EvenMergesort is one of the application of Divide and Conquer Technique. Where the problem is divided into sub problems and find their respective solution. Later combine these solutions to get solution of main problem.



**Correctnessof the algorithm**

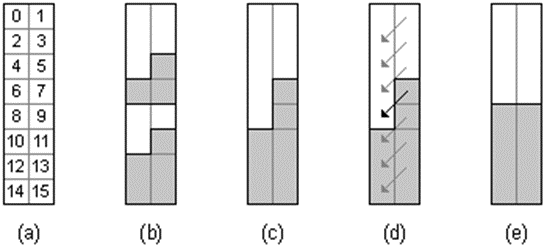
The correctness of the merge algorithm is proved using induction and the [0-1-principle](https://www.inf.hs-flensburg.de/lang/algorithmen/sortieren/networks/nulleinsen.htm).

If *n* = 21 the sequence is sorted by the comparison [0 : 1]. So let *n* = 2*k*, *k* > 1 and assume the algorithm is correct for all smaller *k* (induction hypothesis).

Consider the 0-1-sequence *a* = *a*0, ..., *an*-1 to be arranged in rows of an array with two columns. The corresponding mapping of the index positions is shown in Figure (a), here for *n* = 16.

Then Figure (b) shows a possible situation with a 0-1-sequence.

Each of its two sorted halves starts with some 0's (white) and ends with some 1's (gray).



In the left column the even subsequence is found, i.e. all *ai* with *i* even, namely *a*0, *a*2, *a*4 etc. ;in the right column the odd subsequence is found, i.e. all *ai* with *i* odd, namely *a*1, *a*3, *a*5 etc. Just like the original sequence the even as well as the odd subsequence consists of two sorted halves.

By induction hypothesis, the left and the right column are sorted by recursive application of *odd-even* *merge*(*n*/2) in step 1 of the algorithm. The right column can have at most two more 1's than the left column (Figure (c)).

After performing the comparisons of step 2 of the algorithm (Figure (d)), in each case the array is sorted (Figure (e)

**ALGORITHM**

**Algorithm: *Odd-Even* *Merge*(*n*)**

**INPUT**  : Sequence *a*0, ..., *an*-1 of length *n*>1 whose two halves *a*0, ..., *an*/2-1 and *an*/2, ..., *an*-1 are sorted (*n* a power of 2)

**OUTPUT** : the sorted sequence

1. if *n*>2 then
   1. apply *odd-even* *merge*(*n*/2) recursively to the even subsequence *a*0, *a*2, ...,*an*-2 and to the odd subsequence *a*1, *a*3, , ..., *an*-1
   2. comparison [*i* : *i*+1] for all *i*  {1, 3, 5, 7, ..., *n*-3}
2. else
   1. comparison [0 : 1]

**Algorithm : O*dd-Even* *Mergesort*(*n*)**

**INPUT** : sequence *a*0, ..., *an*-1 (*n* a power of 2)

**OUTPUT** : the sorted sequence

if *n*>1 then

1. apply odd-even mergesort(n/2) recursively to the two halves a0 ,...,an/2-1 and

an/2, ..., an-1 of the sequence

1. *Odd-Even Merge(n)*

**IMPLEMENTATION**

An implementation of odd-even mergesort in Java is given in the following. The algorithm is encapsulated in a class *OddEvenMergeSorter*. Its method *sort* passes the array to be sorted to array *a* and calls function *oddEvenMergeSort*.

Function *oddEvenMergeSort* recursively sorts the two halves of the array. Then it merges the two halves with *oddEvenMerge*.

Function *oddEvenMerge* picks every 2*r*-th element starting from position *lo* and *lo*+*r*, respectively, thus forming the even and the odd subsequence. According to the recursion depth *r* is 1, 2, 4, 8, ....

With the statements

*Sorter s=new OddEvenMergeSorter();*

*s.sort(b);*

an object of type *OddEvenMergeSorter* is created and its method *sort* is called in order to sort array *b*. The length *n* of the array must be a power of 2.

**CODE :**

import java.util.Arrays;

import java.util.Scanner;

interface Sorter

{

void sort(int[] a);

}

public class OddEvenMergesort {

static int[] a ;

public static void main(String[] args)

{

Scanner input = new Scanner(System.in);

System.out.print("Enter Max array size in powers of 2 (i.e 2, 4, 8, 16...) : ");

int n = input.nextInt();

a = new int[n];

System.out.println("Enter the array elements : ");

for(int i=0 ; i <n ;i++)

{

a[i] = input.nextInt();

}

Sorter s = new OddEvenMergeSorter();

s.sort(a);

System.out.println("Sorted Array :");

for(int i = 0;i<n;i++)

System.out.print(a[i]+ " ");

input.close();

}

}

class OddEvenMergeSorter implements Sorter

{

private int[] a;

public void sort(int[] a)

{

this.a=a;

oddEvenMergeSort(0, a.length);

}

/\* \* sorts a piece of length n of the array starting at position low \*/

private void oddEvenMergeSort(int low, int n)

{

if (n>1)

{

int m=n/2;

oddEvenMergeSort(low, m);

oddEvenMergeSort(low+m, m);

oddEvenMerge(low, n, 1);

}

}

/\* \* low is the starting position and n is the length of the piece to be merged,

r is the distance of the elements to be compare \*/

private void oddEvenMerge(int low, int n, int r)

{

int m=r\*2;

if (m<n)

{

oddEvenMerge(low, n, m); // even subsequence

oddEvenMerge(low+r, n, m); // odd subsequence

for (int i=low+r; i+r<low+n; i+=m)

compare(i, i+r);

}

else

compare(low, low+r);

}

private void compare(int i, int j)

{

if (a[i]>a[j])

exchange(i, j);

}

private void exchange(int i, int j)

{

int t=a[i];

a[i]=a[j];

a[j]=t;

}

}

**TIME ANALYSIS**

Let T(n) be the number of comparisons performed by odd-even merge(n).

Then we have for n>2

T(n) = 2·T(n/2) + n/2-1.

With T(2) = 1 we have

T(n) = n/2 · (log(n)-1) + 1 element

.’. T(n) € O(n·log(n)).

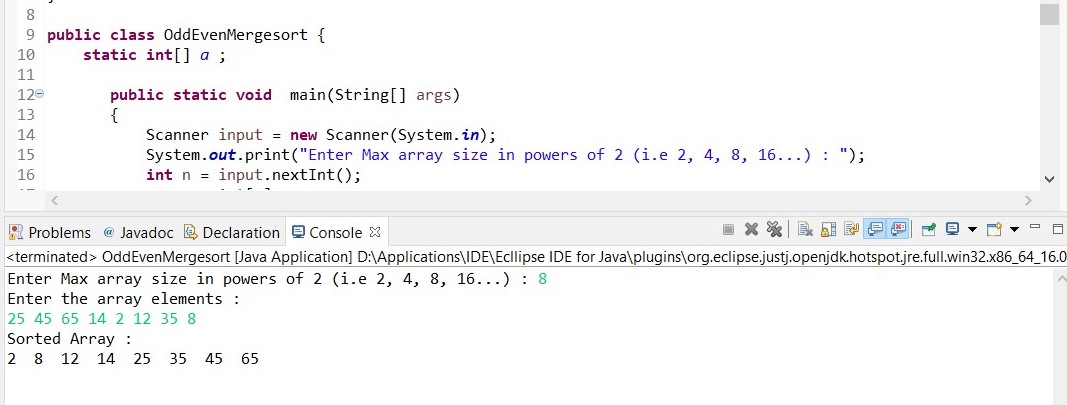
**Time Analysis :**

|  |  |
| --- | --- |
|  |  |
|  |

**APPLICATIONS**

* The single-processor algorithm like bubblesort are generally simple but not very efficient.
* Batcher algorithm is a related but more efficient sort and merge algorithm.
* It uses perfect-shuffle operations and compare-exchange operations.
* It is usually more efficient on parallel processors with long range connections.
* Overcall this algorithm is fast than their counterparts proposed so far.

**OUTPUT**

****