STUDY ON SORTING TECHNIQUES

A PROJECT WRITE-UP

Submitted by

Sudeshna Sur

Under the Supervision of

Prof. Subhashis Majumder

Head of the Department

Computer Science and Engineering

in partial fulfilment for the award of the degree

of

BACHELOR OF TECHNOLOGY

IN

COMPUTER SCIENCE AND ENGINEERING

HERITAGE INSTITUTE OF TECHNOLOGY, KOLKATA

MAY, 2016

ABSTRACT

Let's have a brief introduction with the foundation of a basic sorting algorithm. Then we will gradually move on to expand it further to serve as an improvement over the basic algorithm. The fundamentals of this algorithm are dependent on divide and conquer. However, interesting unlike the traditional quick sort we will be using functions to group the elements of the array.

The first idea of the sort is Level Sort- which will search for elements based on certain **k** hypothetical ranges.

We use the function :- p=(max-min)/k,

where max and min are the largest and smallest of the given number set.

The ranges would be indicated by: R(i)=MIN + (i-1)*p; 1≤i≤k.

The range is obviously between R(i) and R(i+1).

The second part or the improvement of this algorithm is using a modified function:-

$$index(i) = \frac{arr(i) - \min}{max - \min} * k + 1$$

where index(i) is the group assigned to every element arr(i) in the array. The indices are stored in another array and used for placing the related elements side by side thus creating small groups of elements with same indices.

The resulting groups will recursively follow the same process. To speed up the execution time we have used insertion sort at the end for smaller ranges (i.e, for less than 100 elements). In the program we instinctively use k=log(n) for better grouping.

The resulting execution time is evaluated to be faster than the conventional heap sort.

Table of Contents

PART- 1: LEVEL SORT

Objective

We have encountered popular algorithms based on partitioning which involve partitioning a set into 2 subsets. However, it is profitable to create k subsets as a whole in the process to achieve a goal.

Our objective in this article is to give a concept on such a partitioning process where we introduce a sorting algorithm based on variable partitioning. With this variable partitioning concept we introduce a new sort that is almost as fast as quick sort, merge sort and heap sort.

Introduction to Level Sort

The Level Sort attempts to sort an array of elements in non-decreasing order using a systemic strategy of dividing carefully a set of numbers into k subsets in every iteration. The basis of division is dependent on the maximum 'gap' between the numbers i.e, the difference between the maximum and minimum element in the set. If we create predictive ranges between maximum and minimum to create k ranges/ levels, we can narrow down the expectation of finding a suitable number in the ranges attributed. Thus, by suitably assigning numbers in their designated levels we get an almost sorted array. By developing our accuracy in range fixing we ultimately get a completely sorted array. It can almost be imagined as the process by which a density column is formed when immiscible liquids are mixed together.

Foundation Using An Example

Let us imagine an outlay congruent to the set of numbers that are to be sorted. We will try to establish a direct relation between a real life situation and the set of numbers stored in an array A.

A simulator in this case, let it be a cupboard in which numbers are lying in a straight line without any vertical partitions in between the base and the top. Hence the numbers are ordered in any possible permutation of the given length of the array. Suppose, we want to arrange the numbers in a non-decreasing fashion. We have to introduce a distribution of numbers between a certain number of partitions we want to introduce in the cupboard. We choose a number mentally for it to separate numbers of within a range between the vertical partitions. Then, we choose those numbers falling within a certain range would go into a determined level.

We use the formula $p = \frac{MAX-MIN}{k}$;

k=number of partitions;

MAX=maximum of the set of numbers within a range and MIN=minimum of the set of numbers within given range. Using that formula, we gets a set of ranges. We checks for

numbers falling in a specific range and places the numbers in the range indicated by the partitions. The ranges can be calculated easily using the formula:-

$R(i)=MIN + (i-1)*p; 1 \leq i \leq k.$

The range is obviously between R(i) and R(i+1).
Having separated the numbers, the set is almost sorted to a certain degree.
Now, we want to again repeat the process and between the already placed partitions we place k partitions inside keeping in mind that the **number of partitions does not exceed the number of elements inside a level. Then the number of partitions becomes the number of elements in the current level minus 1.**

We do this process for every level we had made by separating the numbers. At, one point we realize that there are 2 numbers left together in a certain level. Thus, by introducing a single partition between them and placing them in non-decreasing order, we gets a cleanly sorted subset within the current outer level.
We go to the outer level's neighbours and do the same. Ultimately, we find that the set of numbers are sorted after a number of repetitions for all levels.

We also use a condition that if MAX = MIN there is no need to introduce any further partitions inside any level.

We then remove all the vertical partitions. The numbers are now sorted in non-decreasing order.

Algorithm

```
1.void LEVEL_SORT(int *A,int s,int e,int k)//k>=2 and <data type here is int>
2. min=MIN(A,s,e);
3. max=MAX(A,s,e);
4.if(max==min || e<=s)
5.  return;
6. k=k<(e-s)?k:(e-s);
7.p=(double)(max-min)/k;
8.for(j=1;j<=k;j++)//loop for partitioning
9.   good=s-1;
10.  a=(double)(min+(j-1)*p);
11.  b=(double)(min+j*p);
12.  flag=false;
13.  if(b>max)
14.   b=max;
15.    for(i=s;i<=e;i++)
16.     if(A[i]>=a && A[i]<b)
17.        flag=true;
18.        temp=A[good+1];
19.        A[good+1]=A[i];
20.        A[i]=temp;
21.        good++;
22.       end if
23.      end inner for
24.   LEVEL_SORT(A,s,good,k);
25.   if(flag==false)//level miss
26.   continue;
27.   s=good+1;
28. return;
```

Choice of k

k should always be greater than equal to 2.
1>k cannot be any negative number as there cannot be negative number of partitions in an array.
2>k cannot be zero as division by k would raise an Arithmetic Exception as well as there cannot be 0 partitions.
3>k cannot be 1 as that would imply division of the superset into 1 set. It is meaningless in this context as the strategy of smart division is not possible here.
4>Any input for k>N implies that k value is virtually rejected and algorithm reduces to hardcore division of array into N-1 partitions. It dramatically increases execution time. That is expectedly the worst case.

Since, the choice of k is up to the user, it must be a general practice to use k as a small value. The execution time "generally" increases with increase of k value. Actually, it is hard to predict which k-value can give us the fastest output. It depends on the topography of numbers, since if there is a "level miss", it means that other levels might potentially have longer depths of recursion if there are different numbers, and at the same time if there are same values in the level then recursion is terminated after a few times. If there is no level miss, and there is almost equi-proportionate number of values lying in a level ,there is the possibility of largest depth of recursion for any level lower than what would have been observed with a 2-partitioned sort.

Here we have been given an N element array. For sorting it we do a for loop for k times to individually look into k sublevels for sorting. The inner for loop finds the suitable values between starting and end indices s and e, by using the "good" pointer. Here, good pointer stays and progresses if only i pointer in the for loop (Line 15) finds a suitable value for the level. With the completion of search for suitable values, the process repeats recursively. At, one point when k increases the number of elements in subset, the number of partitions reduces to the number of elements in subset minus 1.
Also, an important conclusion that if maximum element is equal to minimum element we can skip any further divisions as the subset now consists of equal numbers.

Illustration:- k=3,

34	-2	-5	11	8	0	18	22

Ranges :- -5 to 8, 8 to 21, 21 to 34

-2	-5	0	11	8	34	18	22

s=0 ,e=2 k=k<(e-s)?k:(e-s)=(3 < 2) ? 3 : 2 = 2
So, ranges:- -5 to -2.5 , -2.5 to 0

-5	-2	0	11	8	34	18	22

The first part is sorted.

-5	-2	0	11	8	18	34	22

s=3 e=5 k=k<(e-s) ? k : (e-s) = (3<2) ? 3 :2= 2
So, ranges:- 8 to 13 , 13 to 18

-5	-2	0	8	11	18	34	22

Now, the second part is sorted.

-5	-2	0	8	11	18	34	22

s=6 e=7 k=k<(e-s)?k:(e-s)=(3<2)?3:2=2

So, ranges:- 22 to 28, 28 to 34

-5	-2	0	8	11	18	22	34

The array is now sorted.

TIME COMPARISON

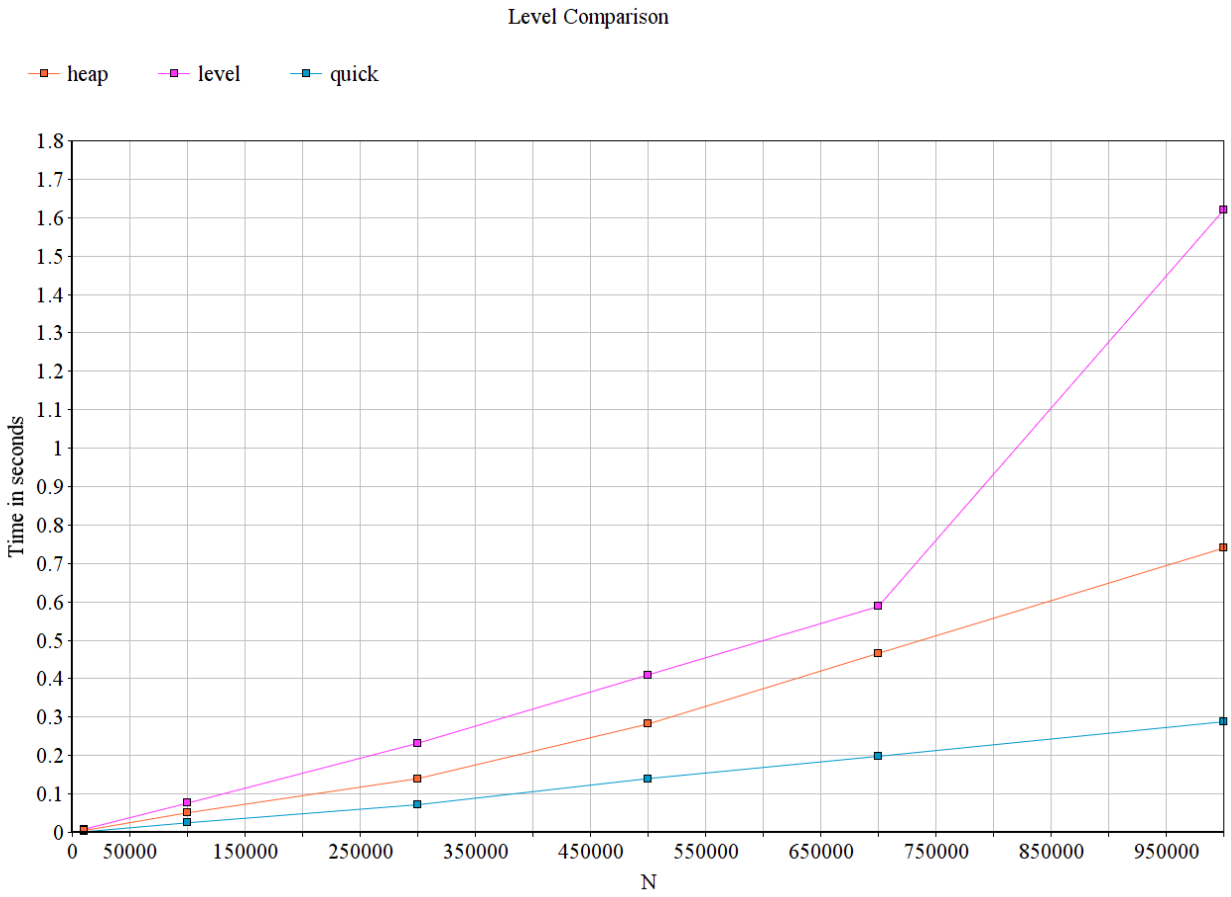
A time comparison for the execution of the respective popularly used sorting algorithms has been compared with Level Sort. The comparison has been done over 5 sets of pseudo-random numbers.
The designated time has been written down in secs:-

N	HEAP	QUICK	LEVEL		
			K=3	K=5	K=7
1,000	0	0	0	0	0
10,000	0.004	0.000	0.008	0.016	0.024
100,000	0.04	0.028	0.076	0.138	0.208
300,000	0.142	0.074	0.232	0.448	0.658
500,000	0.292	0.14	0.41	0.778	1.14
700,000	0.466	0.198	0.588	1.12	1.64
1,000,000	0.74	0.288	0.852	1.62	1.62

We observe a trend for increase in time for sorting with k, however that doesn't imply that that results will be definite for all kinds of arrays. As said above, time differs with the topography of numbers.
This study has been made strictly with respect to pseudo-random numbers and the results vary widely when numbers form a compact range i.e, this sorting algorithm is works faster for numbers within a close range, as recursion terminates early in these cases. Let us see that through another time comparison when the numbers generated are modulo 100.

N	HEAP	QUICK	LEVEL		
			K=3	K=5	K=7
1,000	0	0	0	0	0
10,000	0.004	0.004	0.004	0.006	0.01
100,000	0.032	0.178	0.024	0.046	0.07
300.000	0.106	1.47	0.066	0.13	0.20
500,000	0.196	4.148	0.114	0.222	0.338
700,000	0.296	8.392	0.158	0.308	0.470
1,000,000	0.434	17.392	0.228	0.444	0.672

Following is a graphical comparison of the time taken by the three respective sorts on arrays of various sizes :



Disadvantages

1. Slower than quick, merge and heap sorts for large ranges and doubles/floating point values.
2. Has to look for maximum and minimum in every iteration.

Advantages

1. Is faster over close integral ranges.
2. Asymptotic time complexity in average case is 'expected' to be Θ(nlogn) but it is hard to derive as cases vary based on repetitions.

Conclusion

Level sort is very fast for close ranges and desirable for arrays with repetitions which is the weakness for quick sort. It highly advantageous in special cases where the variation in values are limited such as sorting room temperatures, speed variations in a moving vehicle, or cases in which ranges are close and decimal precision is within 1 to 2 digits. The study for level sort is open for estimation of it's asymptotic time complexity, which this article does not discuss about. It is also expected that level sort can be improvised to be a more efficient algorithm.

PART- 2: A New Hybrid Sorting Technique—Improved Level Interpolation Sorting Technique

Objective

In the previous level sorting technique we have subdivided into k ranges based on the formula –

We have tried to improve the execution time by reducing the 'searching' time and replacing it with predictive ranges beforehand. It is much faster than the conventional heap sorting technique and almost as fast as merge sort. The idea of the function is to group numbers based on the relative difference between the maximum and minimum elements in the array. This will help us arrange the elements into partially sorted order with every iteration. The function we will using is quite similar to the function used in interpolation search hence the name.

Introduction

We will assign a particular index value to each of the array members according to the formula index(i)=$1 + (int)\big((arr[i] - min) * mult\big)$

Where $mult = \frac{(double)k}{max-min}$

We use a particular index(i) array of O(n) since for every array element we will be assigning the "index" value. On the basis of that we will get k+1 subgroups/ levels. With this "partially sorted group" we will follow the process recursively for the current range of similar index(i) values. In the following example we will show the value of k chosen to 3. However in the program we will instinctively use k=log(n)/2 for better execution time as it provides **optimum** number of subgroups.

Let us look at the following example for clearer understanding:-

The groups can be shown in the following way.

Arr	56	32	65	37	55	60	40	12	77
index	3	1	3	2	2	3	2	1	4

1) We take the indices of a similar type and put the elements side by side. We also calculate the start and end indices of each group of elements with same calculated indices in the array.

Arr	32	12	37	55	40	56	65	60	77
INDEX	1	1	2	2	2	3	3	3	4

2) Taking the array elements of one group we repeat the above process.

ARR	12	32	37	55	40	56	65	60	77
INDEX	1	4	2	2	2	3	3	3	4

3) Now the previous subgroup is sorted. We now move on to the next subgroup for which the newly calculated indices are indicated in red in our table given below.

ARR	12	32	37	40	55	56	65	60	77
INDEX			1	1	4				

4) Ignoring the element 55 which is already in its correct position in the sorted subgroup, we perform similar sorting action on the other two elements.

ARR	12	32	37	40	55	56	65	60	77
INDEX			1	4					

5) Next we repeat the process for the subgroup with index value as 3

ARR	12	32	37	40	55	56	65	60	77
INDEX						1	4	2	

6) Ignoring the iteration of 56 we sort the other two elements.

ARR	12	32	37	40	55	56	60	65	77
INDEX							1		4

7) Finally we move on to our last subgroup with index value 4- which, on account of having only one element, is already sorted.

ARR	12	32	37	40	55	56	60	65	77
INDEX									4

Hence, the entire array is sorted.

Assumptions

1) The index array for the current subgroup is indicated in red.

2) Only the current subgroups have been indicated with index values in the later part of the example.

3) In the program we will be using a hybrid sorting technique which will use insertion sort for number of elements less than 100 for a particular group.

4) The value of k used in the program used by us is log(n)/2.

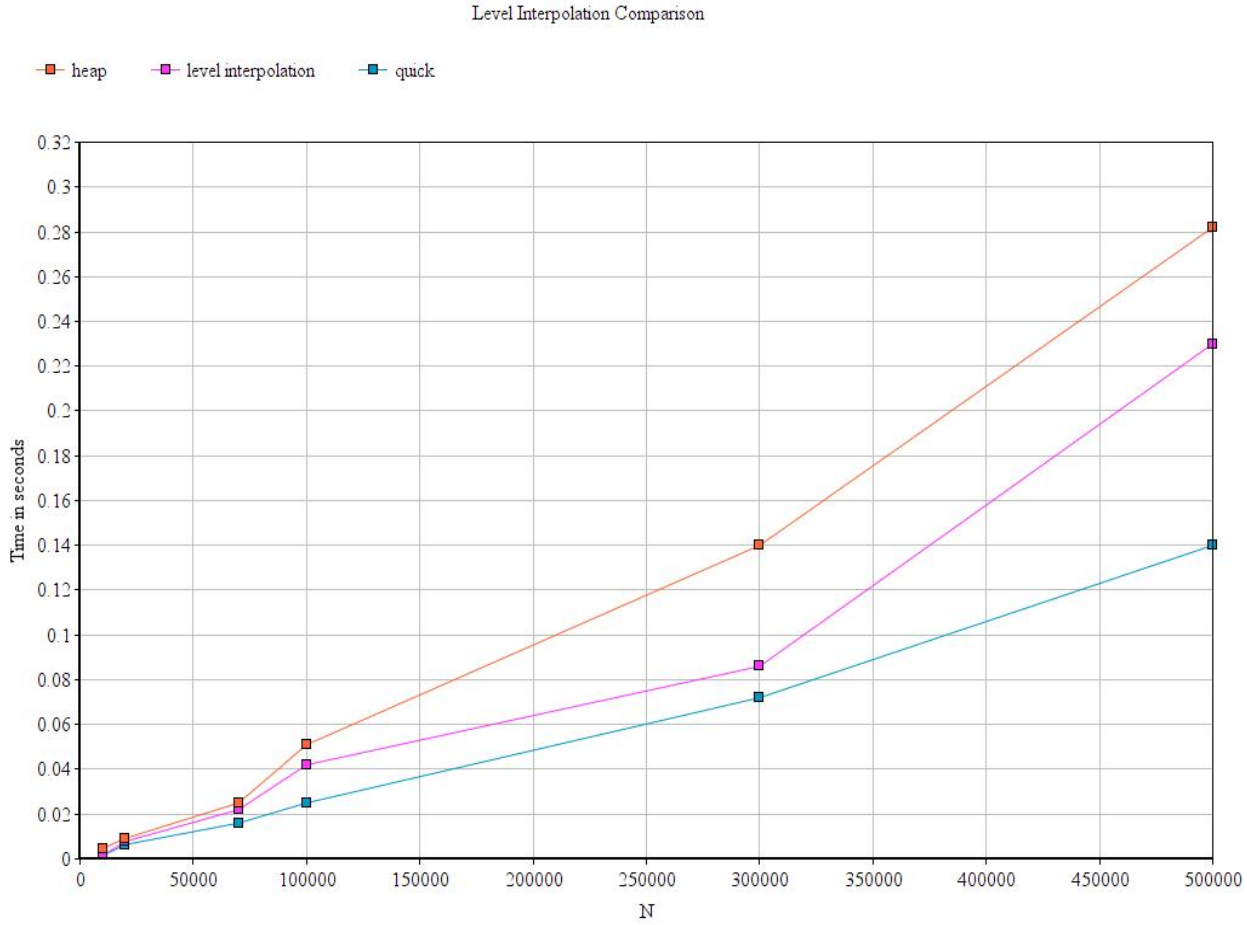
Pseudo-code

```
void function(int *arr,int s,int e,int k)
{

   if(min>=max || e<=s)
   return;
   mult=(double)k/(max-min);// multiplier

   for(i=s;i<=e;i++)
  {
     index[i]=1+(int)((arr[i]-min)*mult);     //function
   }
   pt=s;
   for(i=1;i<=k+1;i++)
  {  st=pt;
     for(j=s;j<=e;j++)  //find elements in a particular k
     {
        if(index[j]==i)
        {
            tmp=arr[j];  //swapping
            arr[j]=arr[pt];
            arr[pt]=tmp;
            tmp=index[j];
            index[j]=index[pt];
            index[pt]=tmp;
            pt++;
         }
     }
     if(pt>st){
     if((pt-1-st)>100)
{
      function(arr,st,pt-1,k);  //use k as log(pt-1-st)/2
}
    else
    insertion(arr,index,st,pt-1);
   }
  }
   return;
  }
```

Time Analysis



Results and Analysis

It is 15-30 % faster than heapsort and 40-80% faster than simple level sort. The improvement over heap and simple level sort becomes more prominent with increase in numbers.

However, it is dependent on an additional array of O(n) to store the function values.