A note on distributions that are sorted faster than others by our new sorting algorithm

Suman Kumar Sourabh
University Department of Statistics
And Computer Applications
T. M. Bhagalpur University
Bhagalpur
India
Email:sourabh.suman@rediffmail.com

Kiran Kumar Sundararjan
Consulting and Analytics
Hexaware Technologies Limited
Navi Mumbai-400710
India
email:kirankumars@hexaware.com

Soubhik Chakraborty*
Department of Applied Mathematics
B. I. T. Mesra
Ranchi-835215
India
***email ID of communicating author: soubhikc@yahoo.co.in**

ABSTRACT

The present paper compares the average complexity (Sorting Time) for different distribution inputs using our new sorting algorithm[1]. We next propose to implement this algorithm by using random pivots and remove the auxiliary array.

KEY WORDS

New Sorting Algorithm, average complexity, comparisons, distribution inputs, random pivots, auxiliary array.

Section I is the introduction, section II supplies the empirical results. Section III is the conclusion and suggestions for future work.

Section I
Introduction

In ref[1] Sundararajan and Chakraborty introduced a new sorting algorithm. We brief it again in appendix A for the benefit of the reader. We straight go to section II for the experimental results. The codes used are given in appendix B.

SECTION – II
Empirical Results

Table – 1: showing mean sorted time (in seconds) to sort *N* numbers from different distribution inputs (averaged over 10 readings)

Table 1

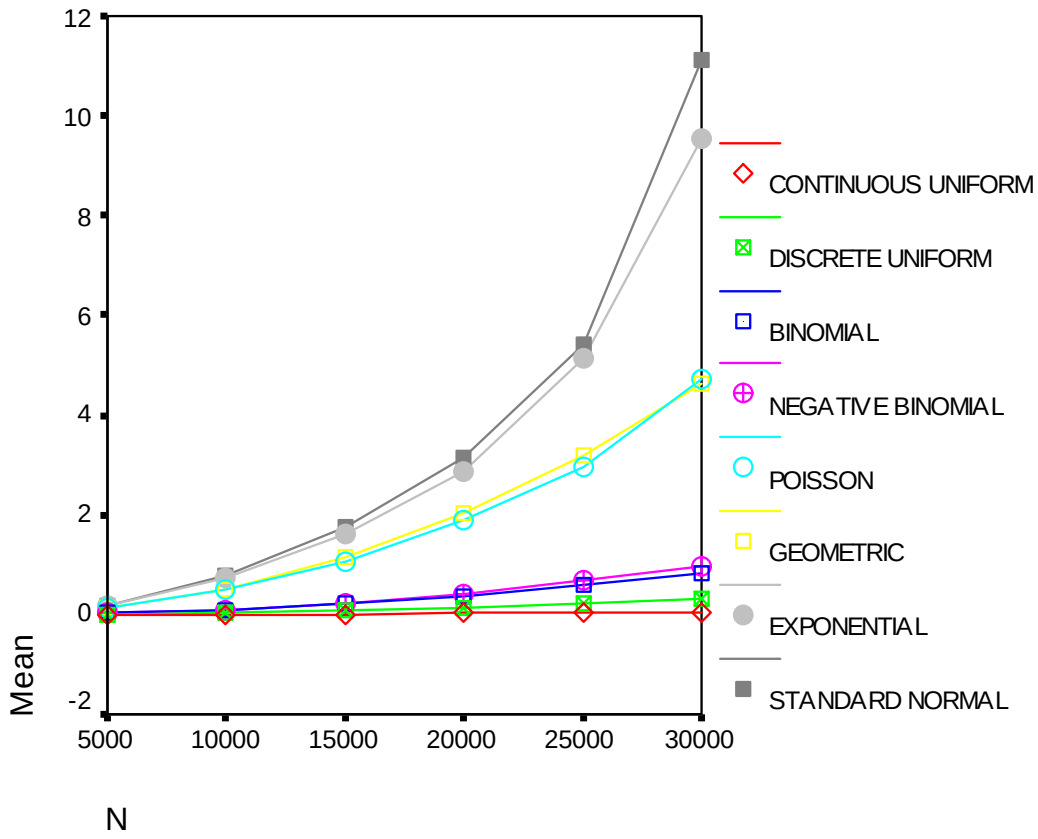
N	*NLogN*	*Binomial (m, p) m=50, p=0.5*	*Discrete Uniform [1,2…k] k=50*	*Geometric (p=0.5)*	*Negative Binomial (k, p) k=50 p=0.5*	*Poisson λ = 1*	*Continuous Uniform [0,1]*	*Exponential [mean = 1]*	*Standard Normal*
5000	42585.966	0.025	0.0101	0.1341	0.0261	0.1241	0.003	0.1811	0.1961
10000	92103.4037	0.0952	0.0391	0.5146	0.1083	0.4838	0.008	0.721	0.7804
15000	144237.082	0.2074	0.0801	1.1657	0.2424	1.0826	0.015	1.6003	1.7334
20000	198069.751	0.3695	0.141	2.055	0.4327	1.9038	0.018	2.8652	3.1453
25000	253165.778	0.5817	0.2173	3.1736	0.676	2.9794	0.0281	5.1603	5.427
30000	309268.58	0.8252	0.3052	4.6214	0.9676	4.7167	0.0323	9.5508	11.111

Table – 2:showing mean sorted time (averaged over 10 readings) in ascending order for a given *N* numbers for different distribution inputs

Table 2

N / *DISTRIBUTIONS*	*5000*	*10000*	*15000*	*20000*	*25000*	*30000*
Continuous Uniform [0,1]	0.003	0.008	0.015	0.018	0.0281	0.0323
Discrete Uniform [1,2…k] k=50	0.0101	0.0391	0.0801	0.1410	0.2173	0.3052
Binomial (m, p) m=50, p=0.5	0.0250	0.0952	0.2074	0.3695	0.5817	0.8252
Negative Binomial (k, p) k=50, p=0.5	0.0261	0.1083	0.2424	0.4327	0.6760	0.9676
Poisson λ = 1	0.1241	0.4838	1.0826	1.9038	2.9794	4.7167
Geometric (p=0.5)	0.1341	0.5146	1.1657	2.0550	3.1736	4.6214
Exponential [mean = 1]	0.1811	0.7210	1.6003	2.8652	5.1603	9.5508
Standard Normal	0.1961	0.7804	1.7334	3.1453	5.4270	11.1110

Mean Time of N from different distribution inputs



SECTION III

CONCLUSION AND SUGGESTIONS FOR FUTURE WORK:

It is clear from the table that inputs from continuous uniform are sorted faster while that from standard normal input is sorted slowest.

The relative orders of the others are as in the table in the descending order of speed i.e. Continuous Uniform < Discrete Uniform < Binomial < Negative Binomial < Poisson < Geometric < Exponential < Standard Normal. We next propose to implement this algorithm by using random pivots and remove the auxiliary array.

System Specifications

Intel Celeron 466 MHz, 128 MB RAM, 80 GB HDD.

APPENDIX - A

1. New Sorting Algorithm (see also ref[1]):

Step 1: Initialize the first element of the array as a pivot (key) element.

Step 2: Starting from the second element, compare it to the pivot element.

Step 2.1: If pivot < element then place the element in the last unfilled position of a temporary array (of same size as the original one).

Step 2.2: If pivot >= element then place the element in the first unfilled position of the temporary array.

Step 3: Repeat step 2 till last element of the array.

Step 4: Finally place the pivot element in the blank position of the temporary array.

(remark: the blank position is created because one element of the original array was taken out as pivot)

Step 5: Split the array into two, based on the pivot element's position.

Step 6: Repeat steps 1-5 till the array is sorted fully.

2. Pseudo code:

```
My_sort(int numbers[], int b[],int l, int r)
      {
       // pivot=first element of array, low=l, up=r ,numbers[] is the
       // original array, b[] is temp array.
       // l is left index and r is right index
       // My logic starts here.
       for (i=l+1; i<=r; i++)
            {
             if (numbers[i] <= pivot)
                  {
                   b[low] = numbers[i];
                   low++;
                  }
            else
                  {
                   b[up]=numbers[i];
                   up--;
                  }
            }
            b[low]=pivot;
            for (i = l; i <= r; i++)
                  numbers[i]=b[i];
            if (l < up-1)
                  My_sort(b,numbers, l, up-1);
            if(up+1 < r)
                  My_sort(b,numbers, up+1,r);
     }
```

3. New Sorting: Function in Borland ‘C++’ ver 5.02

```
void New_Sort(int a[], int b[], int l, int r)
      {
       int pivot=a[l], low=l, up=r;
       for (int i=l+1; i<=r;i++)
            {
             if (a[i] <= pivot)
                  {
                   b[low] = a[i];

             low++;
                  }
             else
                  {
                   b[up]=a[i];
                   up--;
                  }
            }
       b[low]=pivot;
       for (int i = l; i <= r; i++)
            a[i]=b[i];
       if (l<up-1)
            New_Sort(a, b, l, up-1);
       if(up+1<r)
            New_Sort(a, b, up+1,r);
      }
```

4. MySort (Calculating elapsed time): Function in Borland ‘C++’ ver 5.02

```
void MySort(int a[], int array_size)
      {
       clock_t start, finish;
       int *b=new int[array_size];
       double duration;
       start = clock();
       New_Sort(a, b, 0, array_size - 1);
       finish = clock();
       duration = (double) (finish-start)/ CLOCKS_PER_SEC;
       cout.precision(4);
       cout.setf(ios::showpoint);
       cout<<"Elapsed time duration is "<< duration;
      }
```

APPENDIX – B

The *main( )* functions in Borland ‘C++’ Ver. 5.02 for extracting data from different Distribution:

1. Binomial Distribution

```
void main()
{
 int m=100,s;
 float p=0.5,r;
 int n;
 randomize();
 cin>>n;
 int *a=new int[n];
 for(int i=0;i<n;i++)
```

```
      {
       s=0;
       for(int j=0;j<m;j++)
            {
             r=(float)rand()/RAND_MAX;
             if(r<p)
                  ++s;
            }
       *(a+i)=s;
      }
 MySort(a,n);
}
```

2. Discrete Uniform Distribution

```
void main()
{
 int n, k=50;
 float r;
 randomize();
 cin>>n;
 int *a=new int[n];
 for(int i=0;i<n;i++)
      {
       r=(float)rand()/RAND_MAX;
       *(a+i)=(int)(k*r)+1;
      }
 MySort(a,n);
}
```

3. Geometric Distribution

```
void main()
{
 int n;
 float p=0.5,r;
 randomize();
 cin>>n;
 int *a=new int[n];
 for(int i=0;i<n;i++)
      {
       r=(float)rand()/RAND_MAX;
       *(a+i)=1+(int)(log(r)/log(1-p));
      }
 MySort(a,n);
}
```

4. Negative Binomial Distribution

```
void main()
{
 int n,k=10,s,c;
 float p=0.5,r;
 randomize();
 cin>>n;
 int*a=new int[n];
 for(int i=0;i<n;i++)
      {
       s=c=0;
       while(s<k)
                  {
                   ++c;
```

```
                    r=(float)rand()/RAND_MAX;
          if(r<p)
                         ++s;
                    }
       *(a+i)=c;
      }
 MySort(a,n);
}
```

5. Poisson Distribution

```
void main()
{
 int n,Lambda=1,x;
 float p,b=(float) exp((-1)*Lambda),r;
 randomize();
 cin>>n;
 int *a=new int[n];
 for(int i=0;i<n;i++)
      {
       p=1.0;
       for(int j=1;j<5000;j++)
            {
             r=(float)rand()/RAND_MAX;
             p=p*r;
             if(p<b)
                  {
                   x=j-1;
                   break;
                  }
            }
       *(a+i)=x;
      }
 MySort(a,n);
}
```

6. Continuous Uniform Distribution

In Continuous case, the continuous uniform distribution inputs generate real numbers. Therefore, the program requires some alteration in the data types being used.

```
void main()
{
 int n;
 randomize();
 cin>>n;
 float *a=new float[n];
 for(int i=0;i<n;i++)
      *(a+i)=(float)rand()/RAND_MAX;

 MySort(a,n);
}
```

7. Exponential Distribution

```
void main()
{
 int n;
 float r;
 randomize();
 cin>>n;
 int *a=new int[n];
```

```
 for(int i=0;i<n;i++)
      {
       r=(float) rand()/RAND_MAX;
       *(a+i)=(-1)*log(r);
      }
 MySort(a,n);
}
```

8. Standard Normal Distribution

```
void main()
{
 int n;
 float u1,u2;
 randomize();
 cin>>n;
 int *a=new int[n];
 for(int i=0;i<n/2;i++)
      {
       u1=(float) rand()/RAND_MAX;
       u2=(float) rand()/RAND_MAX;
       *(a+i)=sqrt((-2)*log(u1)) * cos(8*atan(1)*u2);
       *(a+n/2+i)=sqrt((-2)*log(u1)) * sin(8*atan(1)*u2);
      }
 MySort(a,n);
}
```

Reference:

[1] K. K. Sundararajan and S. Chakraborty, A new sorting algorithm, Jour. of Applied Math. And Compu. (2006) in press; doi: 10.1016/j.amc.2006.10.065