Programs Optimization in GCC compiler

Mohammad Abd-Allatef Al-Shalabi
Department of IT, Sharjah University

ABSTRACT: The optimization is the efficient translation of a higher-level language into the fastest possible machine language [1], in this project I tried to make an optimization on two programs to achieve the best performance.
I applied many optimization levels such as O0, O1, O2, Os, and O3 on the programs and take the results from each level, after that I made a comparison among the results, as we will see.
After that, I tried to optimize these programs manually. The process is done under LINUX environment using nodes 5 and 6.
KEYWORDS: High Performance, Optimization, Compilation Time.

Introduction

The goal of the optimization compiler is the efficient translation of a higher-level language into the fastest possible machine language that accurately represents the high-level language source. [DS98]

An optimizing compiler usually means simplifying the code, throwing out extraneous instructions, and sharing intermediate results between statements. More advanced optimizations seek to restructure the program and may actually make the code grow in size, though the number of instructions executed will shrink.[DS98]

Before the optimization pass of the compiler process, the compiler must translate the program after syntax and semantic analysis into an intermediate language which typically expressed as a stream of *quadruples* which are statements with exactly one operator and up to two operands.

Levels of optimization:

We explore the optimization levels provided by the GCC compiler, including the specific optimizations provided in each. A large variety of

optimizations are provided by GCC. Some optimizations reduce the size of the resulting machine code, while others try to create code that is faster, potentially increasing its size. [lin**]

- Level zero:

It is the default optimization which provides no optimization at all, this can be explicitly specified with option –O or –O0. [lin**]

- Level 1 (-O1):

The purpose of this level is to produce an optimized image in a short amount of time. These optimizations typically don't require significant amounts of compile time to complete. Level 1 also has two sometimes-conflicting goals. These goals are to reduce the size of the compiled code while increasing its performance. The first level of optimization is enabled as: [lin**]

```
gcc –O1 –o test test.c
```

- Level 2 (-O2):

The second level of optimization performs all other supported optimizations within the given architecture that do not involve a space-speed trade off, a balance between the two objectives. For example, a loop unrolling and function inlining, which have the effect of increasing code size while also potentially making the code faster, are not performed. The second level is enable as:[2]

```
gcc –O2 –o test test.c
```

- Level 2.5 (-Os):

The special optimization level (-Os or size) enables all –O2 optimizations that do not increase code size, it puts the emphasis on size over speed. This includes all second-level optimizations, except for the alignment optimizations, loops, jumps and labels to an address that is a multiple of a power of two, in an architecture-dependent manner. Skipping to these boundaries can increase performance as well as the size of the resulting code and data space, these particular optimizations are disabled. The size optimization level is enabled as: [lin**]

```
gcc –Os –o test test.c
```

- Level 3 (-O3):

The third and highest level enables even more optimizations by putting emphasis on speed over size. This includes optimizations enabled at –O2

and rename-register. The optimization inline-functions also is enabled here, which can increase performance but also can drastically increase the size of the object, depending upon the functions that are inlined. The third level is enabled as: [lin**]

```
gcc  -O3 test test.c
```

Although –O3 can produce fast code, the increase in the size of the image can have adverse effects on its speed. For example, if the size of the image exceeds the size of the available instruction cache, severe performance penalties can be observed. Therefore, it may be better simply to compile at –O2 to increase the chances that the image fits in the instruction cache. [lin**]

Classical optimizations:

- Copy propagation.
- Constant folding.
- Dead code removal.
- Strength reduction.
- Variable renaming.
- Common sub expression elimination.
- Loop-invariant code motion.
- Induction variable simplification.

1. The Process

I take two programs from two students in the class, to apply the optimization level on the two programs.

The first program is used to initially store a random numbers in one dimensional array and then sort these values and compute the sum of these elements, finally it calculate the factorial, Fibonacci for a given number. The first program is shown in Appendix A.

This program contains the following function:

- **sort:** which sort the values inside the array.
- **sum:** which compute the sum of the values inside the array.
- **printArray:** which print the values inside the array.
- **Fibonacci:** compute the Fibonacci for a given number.
- **Factorial:** compute the factorial for a given number.

The second uses some computations inside a nested loop using the factorial, power and gcd functions for a given numbers. The first program is shown in Appendix B.

This program using the following recursion functions:

- **gcd:** which computes the greater common divisor.
- **Factorial:** which computes the factorial of a given number.
- **power:** which computes the power x^y.

In this project I made 10 executions for each program in each level of optimization, executed each program with default optimization then executed each one with optimization levels O0, O1, O2, Os, and O3.

After that, I made manual optimization on each program, and executed each one with the default compiler optimization10 times, and then I recorded the results and analyze them.

The results of the optimization levels of the two programs are shown in the Tables below.

1.1 The results of the first program:

Table 1.1. The result of the default optimization

Default Optimization level			
Test	**Real Time**	**User Time**	**System Time**
1	241.112	241.005	0.03
2	241.564	240.650	0.02
3	241.145	240.214	0.05
4	242.212	241.045	0.17
5	241.125	240.205	0.01
6	242.465	241.567	0.20
7	243.004	239.984	0.01
8	242.992	240.963	0.20
9	241.548	240.021	0.02
10	242.024	241.947	0.01
Average	242.058	240.9655	0.105
Maximum	243.004	241.947	0.20
Minimum	241.112	239.984	0.01

Table 1.2. The result of O1 optimization

O1 optimization level			
Test	**Real Time**	**User Time**	**System Time**
1	87.125	86.493	0.02
2	87.254	87.841	0.01
3	86.483	86.004	0.0
4	87.910	87.145	0.04
5	87.751	86.975	0.01
6	87.045	86.896	0.04
7	87.045	87.254	0.05
8	86.954	87.570	0.02
9	87.254	87.659	0.043
10	86.342	87.847	0.03
Average	87.0465	86.9297	0.0176
Maximum	87.751	87.847	0.05
Minimum	86.342	86.004	0

Table 1.3. The result of O2 optimization

O2 optimization level			
Test	**Real Time**	**User Time**	**System Time**
1	86.704	85.92	0.01
2	86.963	85.92	0
3	85.643	85.63	0.01
4	85.992	85.72	0
5	87.126	86.44	0.01
6	85.975	85.65	0
7	85.836	85.68	0
8	85.941	86.29	0.02
9	86.947	85.92	0.01
10	86.450	85.57	0.04
Average	86.3845	86.18	0.02
Maximum	87.126	86.44	0.04
Minimum	85.643	85.92	0

Table 1.4. The result of Os optimization

Os optimization level			
Test	**Real Time**	**User Time**	**System Time**
1	118.540	116.124	0.001
2	116.927	115.921	0.02
3	116.325	114.124	0.01
4	119.247	114.147	0.0
5	118.014	114.254	0
6	116.241	114.258	0.02
7	118.124	116.122	0.01
8	117.254	114.048	0.00
9	118.124	116.124	0.06
10	119.144	114.14	0.00
Average	117.774	115.124	0.03
Maximum	119.247	116.124	0.06
Minimum	116.241	114.124	0

Table 1.5. The result of O3 optimization

O3 optimization level			
Test	**Real Time**	**User Time**	**System Time**
1	86.074	85.912	0.0
2	86.198	85.142	0.01
3	88.098	84.471	0.01
4	88.147	85.231	0.01
5	85.458	85.1147	0.0
6	85.321	85.265	0.05
7	88.524	86.124	0.01
8	88.486	86.124	0.07
9	88.147	84.545	0.1
10	87.147	85.985	0.8
Average	86.9225	85.2975	0.35
Maximum	88.524	86.142	0.7
Minimum	85.321	84.471	0

From the tables above, we can see that the O2 optimization level is the best one because the average in both real and user time is the lowest one, but the O3 optimization level is greater than O2 and Os, because the O3 optimization level focuses on the size of the program and regardless the performance. After that, I tried to apply a hand tuning on the program, and I get the following replacement:

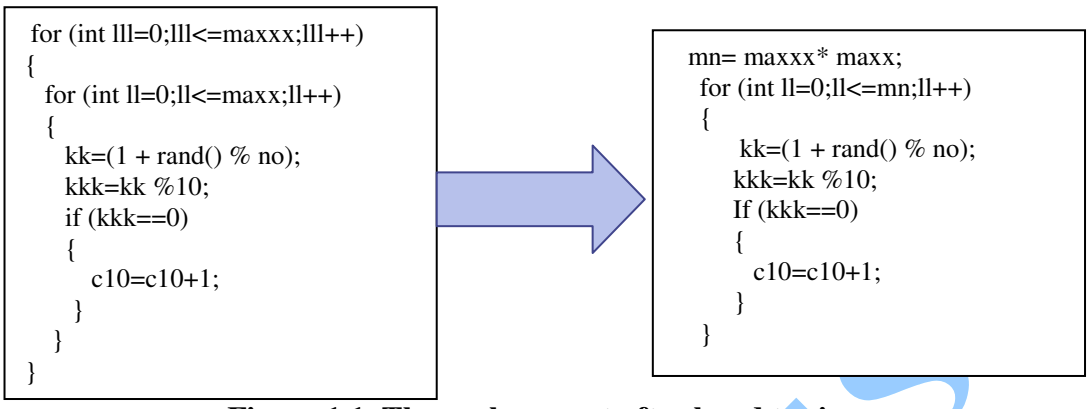


Figure 1.1. The replacement after hand tuning

From the figure above, we can replace the first segment from the code with the second one, means that we can replace the nested for loop with a single one by multiplying the indices. The result of timing the new code is shown in figure 1.6 below.

Table 1.6. The result of hand tuning

Hand Tuning Optimization			
Test	**Real Time**	**User Time**	**System Time**
1	24.015	23.162	0
2	26.124	23.011	0
3	25.897	22.324	0
4	25.821	21.044	0
5	26.145	22.124	0
6	26.012	22.014	0
7	24.124	21.255	0
8	26.145	20.124	0
9	25.014	22.144	0
10	26.475	20.147	0
Average	25.245	21.643	0
Maximum	26.475	23.162	0
Minimum	24.015	20.124	0

We can see the difference from the optimization levels from the figure 1.2 which shows the comparison among the optimization levels and figure 1.3 which show the comparison between O0 and hand tuning optimization.

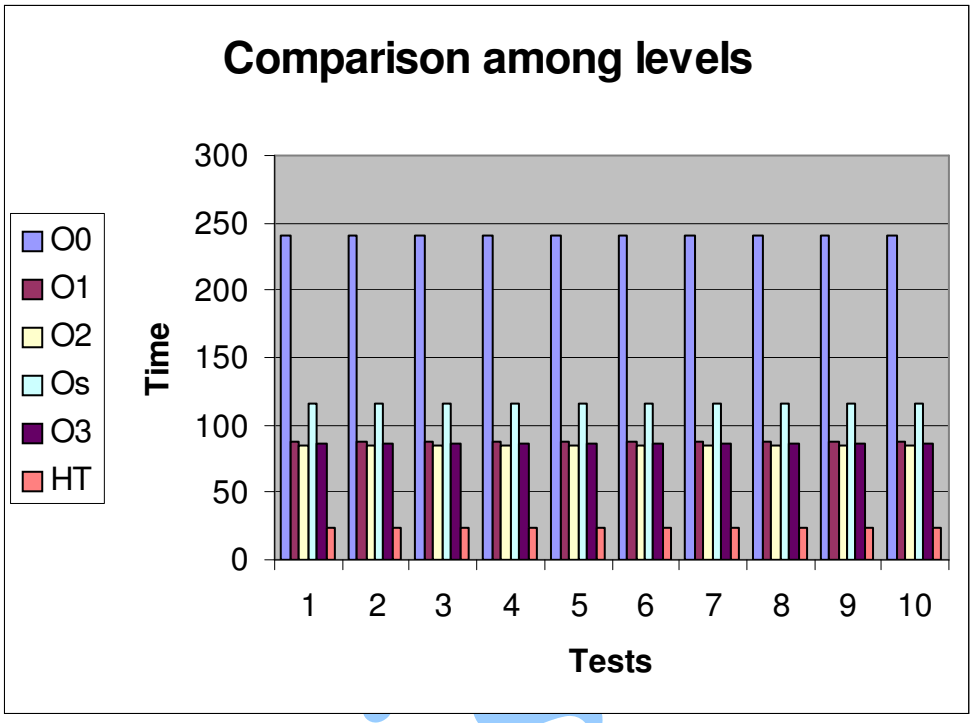


Figure 1.2. Comparison among optimization levels

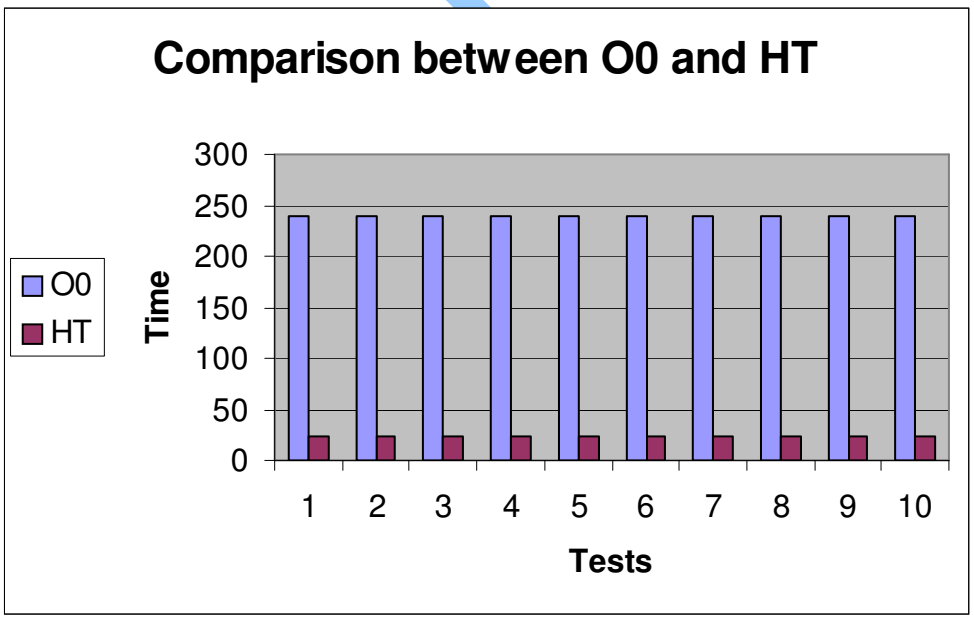


Figure 1.3. Comparison between O0 and hand tuning optimization

We can see from the figures that the hand tuning optimization is the best one, then O2 optimization level, because in the hand tuning we can apply any optimizations on the code that the compiler may not intelligent enough to apply.

1.2 The results of the second program:

Table 1.6. The result of the default optimization

Default Optimization level			
Test	**Real Time**	**User Time**	**System Time**
1	187.372	187.220	0.03
2	187.169	187.160	0.01
3	187.219	187.210	0.0
4	186.220	187.237	0.07
5	187.205	187.200	0.0
6	187.335	187.240	0.06
7	187.183	186.180	0.0
8	187.220	187.212	0.02
9	187.160	187.301	0.0
10	187.098	187.612	0.03
Average	186.796	186.896	0.35
Maximum	187.372	187.612	0.7
Minimum	186.220	186.180	0.0

Table 1.7. The result of O1 optimization

O1 optimization level			
Test	**Real Time**	**User Time**	**System Time**
1	126.156	124.110	1.02
2	126.063	126.060	0.0
3	126.073	126.070	0.0
4	124.856	127.101	0.08
5	126.060	126.060	0.0
6	126.155	123.890	1.01
7	126.048	126.040	0.0
8	126.056	126.055	0.01
9	126.156	124.174	0.09
10	127.134	126.476	0.03
Average	125.995	125.4955	0.51
Maximum	127.134	127.101	1.02
Minimum	124.856	123.890	0

Table 1.8. The result of O2 optimization

O2 optimization level			
Test	**Real Time**	**User Time**	**System Time**
1	133.373	122.960	6.870
2	133.215	133.201	0.02
3	133.241	133.239	0.0
4	135.233	133.301	0
5	133.189	133.240	0.0
6	133.300	133.225	0
7	133.845	132.998	0.03
8	135.152	134.748	0.02
9	133.201	133.241	0.0
10	133.236	133.423	0.04
Average	134.211	128.854	3.435
Maximum	135.233	134.748	6.870
Minimum	133.189	122.960	0

Table 1.9. The result of Os optimization

Os optimization level			
Test	**Real Time**	**User Time**	**System Time**
1	136.314	136.288	0.02
2	136.180	136.200	0.0
3	136.270	136.280	0.0
4	136.311	136.266	0.01
5	136.294	136.290	0
6	135.874	136.214	0.02
7	133.648	137.098	0.05
8	135.812	136.841	0.03
9	136.280	136.315	0.03
10	136.612	136.532	0.01
Average	135.130	136.649	0.025
Maximum	136.612	137.098	0.05
Minimum	133.648	136.200	0

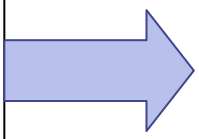
Table 1.10. The result of O3 optimization

O3 Optimization level			
Test	**Real Time**	**User Time**	**System Time**
1	132.313	132.285	0.0
2	132.290	132.193	0.0
3	132.381	132.344	0.0
4	132.312	132.315	0.0
5	132.311	132.310	0.0
6	132.356	132.366	0.0
7	132.255	132.312	0.0
8	132.001	132.801	0.0
9	132.111	132.931	0.0
10	132.496	132.412	0.0
Average	132.2485	132.562	0.0
Maximum	132.496	132.931	0.0
Minimum	132.001	132.193	0.0

From the tables above, we can see that the O1 optimization level is the best one because the average in both real and user time is the lowest one, but the Os optimization level is greater than O2 and O3.

After that, I tried to apply a hand tuning on the program, and I get the following replacement:

```
for  ( int i = 0 ; i < 2000 ; i++ )
{
    for ( int y = 0 ; y <= i ; y++ )
    {
          value += Factorial(y) /
          (double)Power(i,y)+
          ((double)Power(y,i)/Fa
          ctorial(y)) ;
        result += gcd( 1 , 1 , i , y
) ;
     }
     c *= value + result ;
}
```



```
for  ( int i = 0 ; i < 2000 ; i++ )
 {
    for ( int y = 0 ; y <= i ; y++ )
    {
          temp=Factorial(y);
          value += temp /
          (double)Power(i,y)+
          ((double)Power(y,i)/te
          mp) ;
        result += gcd( 1 , 1 , i , y
) ;
     }
```

Figure 1.4. The replacement after hand tuning

From the figure above, we can replace the first segment from the code with the second one, means that we can replace the Factorial(y) statement that redundant twice with a temp variable, means that we reduce the call

statement of the function Factorial from 2 to 1 call statement only, because the call statement take many times. The result of timing the new code is shown in table below.

Table 1.11. The result of hand tuning

Hand Tuning Optimization			
Test	**Real Time**	**User Time**	**System Time**
1	84.015	83.162	0.0
2	86.124	83.011	0.0
3	85.897	82.324	0.0
4	85.821	81.044	0.0
5	86.145	82.124	0.0
6	86.012	82.014	0.0
7	84.124	81.255	0.0
8	86.145	80.124	0.0
9	85.014	82.144	0.0
10	86.475	80.147	0.0
Average	85.245	81.643	0.0
Maximum	86.475	83.162	0.0
Minimum	84.014	80.124	0.0

We can see the difference from the optimization levels from the figure 1.5 which show the comparison among the optimization levels and figure 1.6 which show the comparison between O0 and hand tuning optimization.

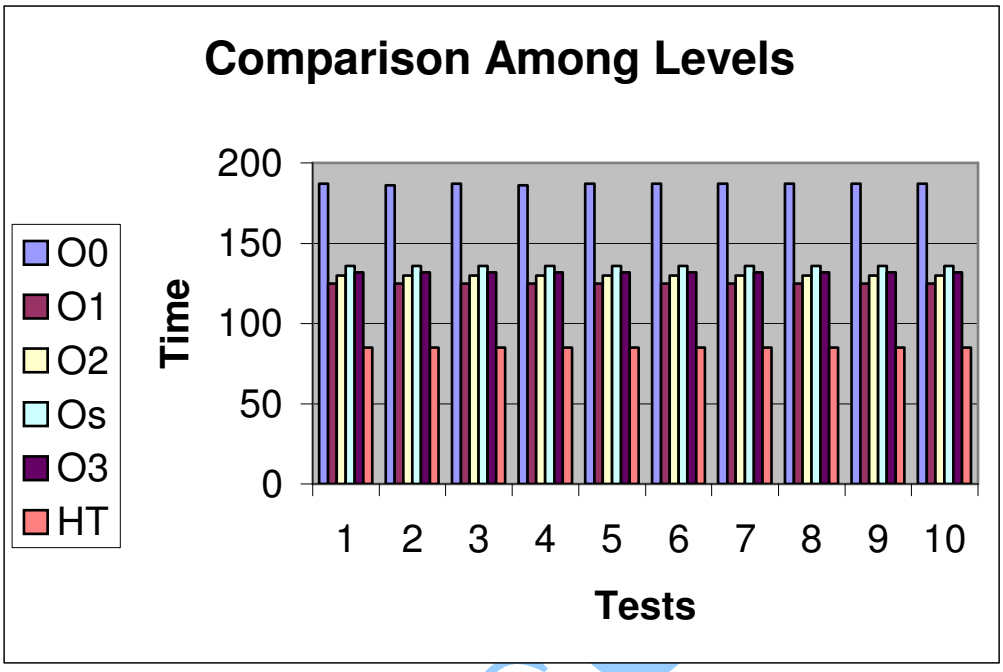


Figure 1.5. Comparison among optimization levels

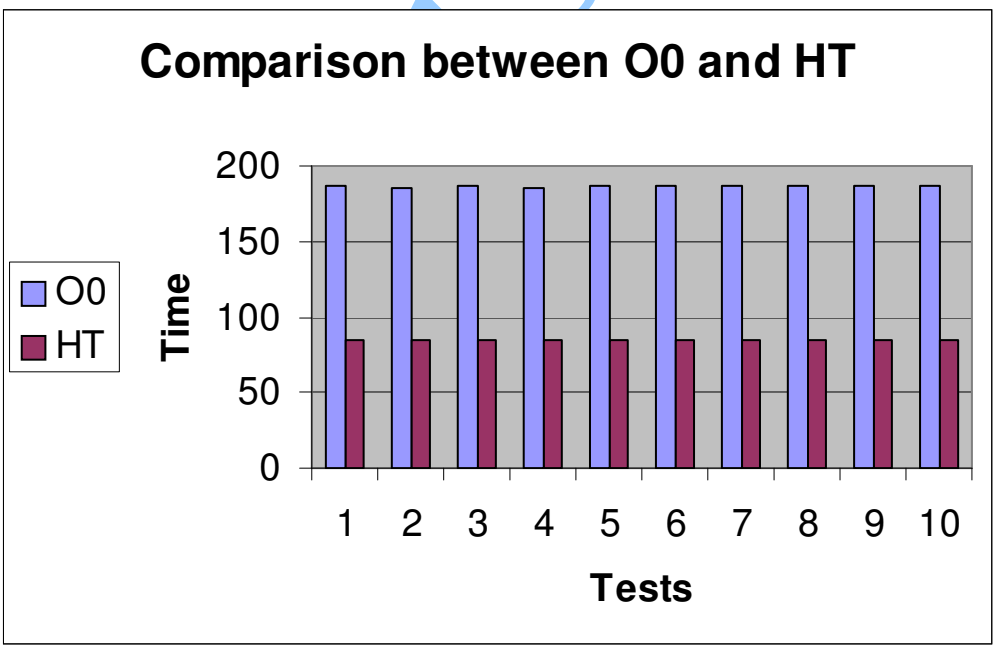


Figure 1.6. Comparison between O0 and hand tuning optimization

We can see from the figures that the hand tuning optimization is the best one, then O1 optimization level, because in the hand tuning we can apply any optimizations on the code that the compiler may not intelligent enough to apply.

Conclusions

1. Optimization is the good way to achieve a best performance of our applications.

2. There are many different optimization levels.

3. Sometimes, the O2 optimization level is the simplest way to achieve a good performance, but not always.

4. Sometimes, the optimization levels don't get the best performance, so we need to optimize our application manually.

References

[DS98] **K. Down, C. Severance** – *High Performance Computing*, Second edition, O'Reilly, 1998.

[KK03] **G. Karniadakis, R. Kirby** – *Parallel Scientific Computing in C++ an MPI*, Cambridge University Press, 2003.

[K+94] **V. Kumar, A. Grama, A. Gupta, G. Karypis** – *Introduction to Parallel Computing,* Benjamin Cummings / Addison Wesley, 1994.

[LS08] **C. Lin, L. Snyder** – *Principles of Parallel Programming*, Pearson Education, 2008.

[lin**] www.linuxjournal.com

Appendix A: The first Program

```
/* this program do the following:
1- find random number between 0 and 32 and find frequency of no.10
2- find random factorial and fibonacci series.
3- put random number in(0,32) in array a.
4- sort array a in array b.
5- find the sum of array elements
*/
#include <iostream.h>
#include <iomanip.h>
#include <stdlib.h>
#define no 32
#define maxx 100000
#define maxxx 10000
unsigned long factorial(unsigned long);
long fibonacci(long);
void Sort(int [] ,int  );
int sum(int [], int);
void printArray(int[], int);
main()
{ int result,reslt,kk,kkk;
          int c10=0;
          int a[no],b[no];
          int i;
           for (int lll=0;lll<=maxxx;lll++){
  for (int ll=0;ll<=maxx;ll++){
          kk=(1 + rand() % no);
          kkk=kk %10;
          if(kkk==0){ c10=c10+1;}}
  }
for (int k=0; k<=no;k++)
{
          for ( i = 0; i <= no; i++)
          {
          a[k]=factorial(i);
          result = fibonacci(i);
          }
}
          cout << "Fibonacci(" << i << ") = " << result << endl;
```

```
            cout << setw(2) << i << "! = " << factorial(i) << endl;
            for (int j = 0; j<= no; j++)
            {a[j]=(1 + rand() % no);
            }
            for (int m = 0; m<= no; m++)
            {
    b[no-m]=a[m];
            }
cout << "array a before sorting"<<endl;
cout<<"================================="<<endl;
   printArray(a, no);
   cout<<endl;
cout << "array a after sorting"<<endl;
cout<<"================================="<<endl;
   printArray(b, no);
   cout<<endl;
    reslt = sum(b, no);
  cout<<"========================="<<endl;
   cout<<"sum of array elements="<<reslt<<endl;
   Sort(b, no);
  cout<<endl;
   cout<<"number 10 freq. "<<c10<<endl;
   return 0;
}
// Recursive definition of function factorial
unsigned long factorial(unsigned long number)
{
   if (number <= 1)
      return 1;
   else
      return number * factorial(number - 1);
}
// Recursive definition of function fibonacci
long fibonacci(long n)
{
   if (n == 0 || n == 1)
      return n;
   else
      return fibonacci(n - 1) + fibonacci(n - 2);
}
int sum(int b[], int size)
```

```
{
 if (size == 1)
  return b[0];
else
 return b[size - 1] + sum(b, size - 1);
}
void Sort(int a[], int size)
{
   int temp;
   for (int pass = 1; pass < size; pass++)
      for (int j = 0; j < size - 1; j++)
         if (a[j] > a[j+1]) {
            temp = a[j];
            a[j] = a[j+1];
            a[j+1] = temp;
         }
}
void printArray(int a[], int size)
{
   for (int l = 0; l < size; l++) {
      if (l % 8 == 0)
         cout << endl;
      cout << setw(5) << a[l];
   }
}
```

Appendix B: The second Program

```
#include<iostream.h>
using namespace std ;
long Power( int b , int p ) ;
long Factorial( long value ) ;
int gcd( int , int , int , int ) ;
int main( )
{
    double value = 0 ;
    double result = 1 ;
    double c = 1 ;
    for ( int i = 0 ; i < 2000 ; i++ )
    {
       for ( int y = 0 ; y <= i ; y++ )
```

```
        {
            value += Factorial(y) / (double)Power(i,y)+((double)Power(y,i)/Factorial(y)) ;
            result += gcd( 1 , 1 , i , y ) ;
        }
        c *= value + result ;
    }
    cout<<c<<endl ;
    return 0 ;
}
int gcd( int c , int g , int x , int y )
{
   if ( (x<c) || (y<c) )
     return g ;
   if ( (x%c==0) && (y%c==0) )
   {
     g = c ;
   }
   gcd( c + 1 , g , x , y ) ;
}
long Power( int b , int p )
{
   if ( p <= 0 )
   {
     return 1 ;
   }
   else
   {
     return ( b * Power( b , p - 1 ) ) ;
   }
}
long Factorial( long value )
{
   if ( value <= 1 )
   {
     return 1 ;
   }
   else
   {
     return ( value * Factorial( value - 1 ) ) ;
   }
}
```