CS 341 - Programming Project 2

Awnon K. Bhowmik

December 9, 2019

Analysis of Searching and Sorting Algorithms

For this project we are to compare the runtimes of different algorithms.

Analysis of searching algorithms

In this section we are to discuss and analyze the two well known searching algorithms **Linear Search**, also known as **Sequential Search** and **Binary Search**.

Linear Search Algorithm

1. Given an array or list of n elements, and a **target element** x.
2. Loop through the entire array one by one.
3. In each step, check whether the number in the list index matches the target element.
4. If a match is found, return the index where it was found
5. If no match found, return an appropriate index resembling it.

Binary Search Algorithm

1. Given an array or list of n elements, and a **target element** x.
2. Pick a central element,

$$\text{Pivot or mid index} = \begin{cases} \dfrac{n}{2} & n = 2k+1, k \in \mathbb{Z} \\ \left\lfloor \dfrac{n}{2} \right\rfloor & n = 2k, k \in \mathbb{Z} \end{cases}$$

3. In each step, check whether the number in the mid index matches the target element.
4. If the target element is to the left of the list at mid index, discard the right half, and work with the remaining left part. If the target element is to the right of the list at mid index, discard the left half, and work with the remaining right part.
5. If a match is found, function returns the mid index.
6. If no match found, return an appropriate index resembling it.

Algorithm to analyze searching algorithm

- In the program, we generate random numbers into a file, of search size 2^n where n ranges from 1 to 22. This was done so we can have sufficient data which can be used to prove some results graphically. It is recommended to reduce this maximum value of n to an appropriate integer if the processor is not strong enough to handle the computational and execution complexity.
- Every time we read a file with a particular file size, we perform linear search and rewind the file using the code

```
        inFile.clear();
        inFile.seekg(0,ios_base::beg);
```

- Record the execution time for every trial, and display it to the console screen. Instead of using clock() method from time.h we decided to use

```
          high_resolution_clock();
```

  method from chrono library
- Exit the process

The C++ code

```
#include<iostream>
#include<fstream>
#include<vector>
#include<algorithm>
#include<chrono>
#include<ctime>
#include<random>
#include<iomanip>
#include<tuple>

using namespace std;
using namespace chrono;

random_device rd;
mt19937_64 mt(rd());
uniform_int_distribution<> dist(0, (int)pow(10, 9));

long long int ls_index;
long long int bs_index;

unsigned long long count_binary = 0;

ifstream inFile;
ofstream outFile;

void fill_File(int size) {
   outFile.open("numbers.txt", ios_base::trunc);

   int i = 0;
   while (i != size) {
      outFile << dist(mt) << "\n";
      i++;
   }
   outFile.close();
}

tuple<long long, unsigned long long>
LinearSearch(vector<unsigned long long>vec, unsigned long long x) {
   long long ls_index = -1;
   unsigned long long count_linear = 0;
   for (unsigned long long i = 0; i < vec.size(); i++) {
      count_linear++;
      if (vec[static_cast<size_t>(i)] == x) {
         ls_index = i;
         break;
      }
   }
   return { ls_index,count_linear };
}

tuple<long long, unsigned long long>
```

```cpp
BinarySearch(vector<unsigned long long>vec,
   unsigned long long low, unsigned long long high, unsigned long long x) {

   while (low < high)
   {
      unsigned long long mid = (low + high) / 2;
      count_binary++;
      if (low <= mid || mid >= high) {
         bs_index = -1;
         return { bs_index,count_binary };
      }
      else if (x > vec[static_cast<size_t>(mid)])
         return BinarySearch(vec, mid + 1, high, x);
      else if (x < vec[static_cast<size_t>(mid)])
         return BinarySearch(vec, low, mid - 1, x);
      else {
         bs_index = mid;
         return { bs_index,count_binary };
      }
   }

   return { -1,count_binary };
}

int main(int argc, char** argv) {
   vector<unsigned long long>a;

   inFile.open("numbers.txt");
   outFile.open("numbers.txt");

   if (!inFile || !outFile) {
      cerr << "Error opening file";
      return 0;
   }

   unsigned long long x, y;

   cout << "Sequential Search\n\nFile Size\tNumber\tIndex\tTime\tNumber of steps\n";

   double SumLinearSearch = 0;
   for (int i = 1; i <= 22; i++) {

      fill_File((int)pow(2, i));

      a.clear();
      while (!inFile.eof()) {
         inFile >> x;
         a.push_back(x);
      }

      //Clear the file and return pointer to beginning of file
      inFile.clear();
      inFile.seekg(0, ios_base::beg);

      y = dist(mt);

      long long ls_index;
      unsigned long long ls_count;

      auto t1 = high_resolution_clock::now();
      tie(ls_index, ls_count) = LinearSearch(a, y);
      auto t2 = high_resolution_clock::now();

      duration<double, milli> diff = t2 - t1;
      SumLinearSearch += diff.count();

      cout << setw(7) << (int)pow(2, i) << setw(15) << y << "\t"
         << setw(3) << ls_index << setw(10) << SumLinearSearch << "\t" << setw(8) <<
            ls_count << "\n";
   }

   cout << "\n\nBinary Search\n\nFile Size\tNumber\tIndex\tTime\tNumber of Steps\n";

   double SumBinarySearch = 0;
   for (int i = 1; i <= 22; i++) {

      fill_File((int)pow(2, i));

      a.clear();
      while (!inFile.eof()) {
         inFile >> x;
         a.push_back(x);
      }

      unsigned long long low = 0;
      unsigned long long high = a.size() - 1;
```

```
        /*Notice that here in Binary Search, the function is written so
        that the program always knows which index to search from, this
        is why we don't need to rewind the file pointer to look for the
        starting and ending point.*/

        //Binary search requires the list to be sorted
        sort(a.begin(), a.end());

        y = dist(mt);

        long long bs_index;
        unsigned long long bs_count;

        auto t3 = high_resolution_clock::now();
        tie(bs_index, bs_count) = BinarySearch(a, low, high, y);
        auto t4 = high_resolution_clock::now();

        duration<double, milli>diff = t4 - t3;
        SumBinarySearch += diff.count();

        cout << setw(7) << (int)pow(2, i) << setw(15) << y << "\t"
           << setw(3) << bs_index << setw(12) << SumBinarySearch << "\t" << setw(5) <<
               bs_count << "\n";
    }

    inFile.close();
    outFile.close();
    return 0;
}
```

Console Output

Although the program gives the outputs in two separate tabular form, here we'll just compile them in the same table, so its easier to compare the execution times side by side. The time is calculated in milliseconds to increase accuracy.

Linear Search					**Binary Search**				
File	Number	Index	Time	Number of Steps	File	Number	Index	Time	Number of Steps
2	9841359	-1	0.0259	1	2	220816134	-1	0.0063	1
4	209185955	-1	0.0322	5	4	65795147	-1	0.016	2
8	833708208	-1	0.0382	9	8	572455325	-1	0.0242	3
16	28128157	-1	0.0439	17	16	771004053	-1	0.0337	4
32	77150541	-1	0.0524	33	32	356949669	-1	0.0417	5
64	318270599	-1	0.0609	65	64	958435947	-1	0.0499	6
128	900729034	-1	0.0726	129	128	990517142	-1	0.0574	7
256	343936249	-1	0.0967	257	256	212704272	-1	0.063	8
512	516716010	-1	0.1258	513	512	711782926	-1	0.0708	9
1024	238693152	-1	0.1807	1025	1024	505372654	-1	0.0764	10
2048	739217550	-1	0.282	2049	2048	747274833	-1	0.0863	11
4096	789134060	-1	0.5639	4097	4096	586656825	-1	0.0916	12
8192	166035152	-1	1.0141	8193	8192	120870138	-1	0.1264	13
16384	878034570	-1	1.8205	16385	16384	163257089	-1	0.132	14
32768	608754666	-1	4.4571	32769	32768	491655710	-1	0.1395	15
65536	694300027	-1	9.5779	65537	65536	143486137	-1	0.1709	16
131072	587529353	-1	17.1452	131073	131072	193100266	-1	0.1765	17
262144	695371951	-1	33.4916	262145	262144	343569789	-1	0.2117	18
524288	746253008	-1	63.2732	524289	524288	701444084	-1	0.2186	19
1048576	719565499	-1	132.486	1048577	1048576	29506686	-1	0.2244	20
2097152	204417884	-1	264.879	2097153	2097152	184314335	-1	0.2303	21
4194304	176233490	-1	533.471	4194305	4194304	991881795	-1	0.2358	22

Table 1: Comparing Binary and Linear Search

If we notice closely, the steps of execution in Linear search is always $(n+1)$ and that of Binary search is $\log_2 n$. So it is proved that the time complexity of these algorithms are $O(n)$ and $O(\log n)$ respectively. This can also be supported with a graphical proof.

Graphical Proof

We make use of *Wolfram Mathematica* to write the following script...

```
In[128]:= data := {{2, 0.0259}, {4, 0.0322}, {8, 0.0382}, {16, 0.0439}, {32, 0.0524}, {64, 0.0609},
   {128, 0.0726}, {256, 0.0967}, {512, 0.1258}, {1024, 0.1807}, {2048, 0.282},
   {4096, 0.5639}, {8192, 1.0141}, {16384, 1.8205}, {32768, 4.4571}, {65536, 9.5779},
   {131072, 17.1452}, {262144, 33.4916}, {524288, 63.2732}, {1048576, 132.486},
   {2097152, 264.879}, {4194304, 533.471}}

In[129]:= lm = Fit[data, {1, x}, x] // Normal

Out[129]= -0.0766146 + 0.000126943 x

In[130]:= a := ListPlot[data, PlotStyle → {RGBColor[1, 0, 0], PointSize[Medium]}]
          b := Plot[lm, {x, 0, 2^22}, PlotRange → All]

In[132]:= Show[b, a, AxesLabel → {"Search Size", "Time"}, PlotLabel → "Linear Search O(n)",
  LabelStyle → Directive[Blue, Bold]]
```

Out[132]=

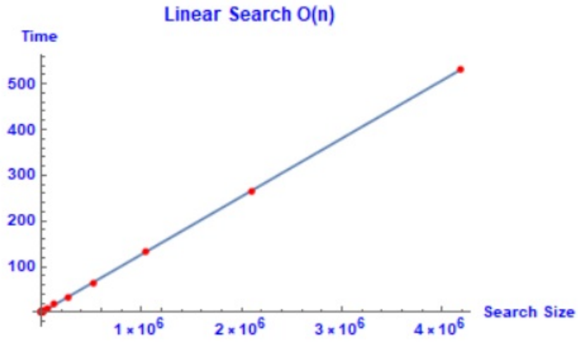

Fig 1. **Linear Search Complexity** $O(n)$

It is seen that the best fit closely resembles

$$y = 0.000126943x - 0.0766146$$

which is a straight line

We repeat this process for the Binary Search algorithm, by writing the following script

```
In[1]:= data := {{2, 0.0063}, {4, 0.016}, {8, 0.0242}, {16, 0.0337}, {32, 0.0417}, {64, 0.0499},
   {128, 0.0574}, {256, 0.063}, {512, 0.0708}, {1024, 0.0764}, {2048, 0.0863},
   {4096, 0.0916}, {8192, 0.1264}, {16384, 0.132}, {32768, 0.1395}, {65536, 0.1709},
   {131072, 0.1765}, {262144, 0.2117}, {5224288, 0.2186}, {1048576, 0.2244},
   {2097153, 0.2303}, {4194304, 0.2358}}

nlm = NonlinearModelFit[data, {a + b * Log[c * x] / Log[2], c > 0}, {a, b, c}, x] // Normal

a := ListPlot[data, PlotStyle → {RGBColor[1, 0, 0], PointSize[Medium]}]
b := Plot[nlm, {x, 0, 2^22}, PlotRange → All]
Show[a, b, AxesLabel → {"Search Size", "Time"}, PlotLabel → "Binary Search O(Log n)",
 LabelStyle → Directive[Blue, Bold]]

ClearAll["Global`*"]

Out[2]= -0.0203193 + 0.0163149 Log[1.09271 x]
```

Out[5]=

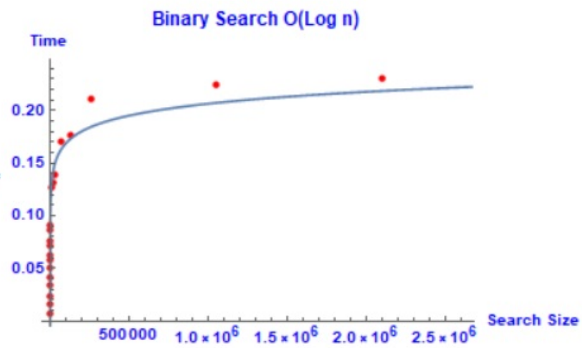

Fig 2. **Binary Search Complexity** $O(\log n)$

Analysis of sorting algorithms

In this section we will discuss and analyze a few well known sorting algorithms, and prove their computational complexity. It includes

- Bubble Sort $O(n^2)$
- Insertion Sort $O(n^2)$
- Merge Sort $O(n \log n)$

We will once again generate a list of random numbers in files, and sort them, starting from a file size of 500 records and going upto 10,000 in steps of 500.

The C++ code

```
#include<iostream>
#include<vector>
#include<algorithm>
#include<fstream>
#include<random>
#include<chrono>
#include<iomanip>

using namespace std;
using namespace chrono;

ifstream inFile;
ofstream outFile;

random_device rd;
mt19937_64 mt(rd());
uniform_int_distribution<>dist(0, 1000);

vector<unsigned long long>a;

void fill_File(int size) {
   outFile.open("numbers.txt", ios_base::trunc);

   int i = 0;
   while (i != size) {
      outFile << dist(mt) << "\n";
      i++;
   }
   outFile.close();
}

void reset() {
   //clear the file and return pointer to the begining of file
   inFile.clear();
   inFile.seekg(0, ios_base::beg);
}

void BubbleSort(vector<unsigned long long> &vec) {
   for (unsigned long long i = 0; i < vec.size(); i++) {
      for (unsigned long long j = i + 1; j < vec.size(); j++) {
         if (vec[static_cast<size_t>(j)] > vec[static_cast<size_t>(i)])
            swap(vec[static_cast<size_t>(j)], vec[static_cast<size_t>(i)]);
      }
   }
}

void InsertionSort(vector<unsigned long long> &vec) {
   unsigned long long key;

   for (unsigned long long i = 1; i < vec.size(); i++) {
      key = vec[static_cast<size_t>(i)];
      unsigned long long j = i;

      while (static_cast<size_t>(j) > 0 && vec[static_cast<size_t>(j-1)] > key) {
         vec[static_cast<size_t>(j)] = vec[static_cast<size_t>(j-1)];
         j--;
      }
      vec[static_cast<size_t>(j)] = key;
   }
}

void Merge(vector<unsigned long long>& vec,
```

```
    unsigned long long low, unsigned long long mid, unsigned long long high) {
    //split from the middle
    vector<unsigned long long>leftVec(static_cast<size_t>(mid - low + 1));
    vector<unsigned long long>rightVec(static_cast<size_t>(high - mid));

    //fill in the left list
    for (size_t i = 0; i < leftVec.size(); i++)
        leftVec[i] = vec[(size_t)(low + i)];

    //fill in the right list
    for (size_t i = 0; i < rightVec.size(); i++)
        rightVec[i] = vec[(size_t)(mid + 1 + i)];

    // initial indexes of first and second subarrays
    unsigned long long leftIndex = 0, rightIndex = 0;

    // the index we will start at when adding the subarrays back into the main array
    unsigned long long currentIndex = low;

    // compare each index of the subarrays adding the lowest value to the currentIndex
    while (leftIndex < leftVec.size() && rightIndex < rightVec.size()) {
        if (leftVec[(size_t)(leftIndex)] <= rightVec[(size_t)(rightIndex)]) {
            vec[(size_t)(currentIndex)] = leftVec[(size_t)(leftIndex)];
            leftIndex++;
        }
        else {
            vec[(size_t)(currentIndex)] = rightVec[(size_t)(rightIndex)];
            rightIndex++;
        }
        currentIndex++;
    }

    // copy remaining elements of leftArray[] if any
    while (leftIndex < leftVec.size()) {
        vec[(size_t)(currentIndex)] = leftVec[(size_t)(leftIndex)];
        currentIndex++;
        leftIndex++;
    }

    // copy remaining elements of rightArray[] if any
    while (rightIndex < rightVec.size()) {
        vec[(size_t)(currentIndex)] = rightVec[(size_t)(rightIndex)];
        currentIndex++;
        rightIndex++;
    }
}

void MergeSort(vector<unsigned long long>& vec,
    unsigned long long low, unsigned long long high) {

    //base case
    if (low < high) {
        unsigned long long mid = (low + high) / 2;

        //sort left list
        MergeSort(vec, low, mid);

        //sort right list
        MergeSort(vec, mid + 1, high);

        //merge the lists
        Merge(vec, low, mid, high);
    }
}

int main() {

    inFile.open("numbers.txt");
    outFile.open("numbers.txt");

    if (!inFile || !outFile) {
        cerr << "Error opening file!";
        return 0;
    }

    unsigned long long x;

    cout << "\t\t\t\tRuntime Analysis\n\nFile Size\tBubble Sort\tInsertion Sort\tMerge
        Sort\n";

    double sumBubbleTime = 0;
    double sumInsertionTime = 0;
    double sumMergeTime = 0;

    for (int i = 500; i < 10500; i += 500) {
```

```
        //fill the file with random numbers
        fill_File(i);

        //Clear the vector to avoid errors. Then, copy the file contents into the vector.
        a.clear();
        a.shrink_to_fit();
        while (!inFile.eof()) {
           inFile >> x;
           a.push_back(x);
        }

        auto t1 = high_resolution_clock::now();
        BubbleSort(a);
        auto t2 = high_resolution_clock::now();

        duration<double, milli>BubbleSortTime = t2 - t1;
        sumBubbleTime += BubbleSortTime.count();

        reset();

        //Clear the vector to avoid errors. Then, copy the file contents into the vector.
        a.clear();
        a.shrink_to_fit();
        while (!inFile.eof()) {
           inFile >> x;
           a.push_back(x);
        }

        auto t3 = high_resolution_clock::now();
        InsertionSort(a);
        auto t4 = high_resolution_clock::now();

        duration<double, milli>InsertionSortTime = t4 - t3;
        sumInsertionTime += InsertionSortTime.count();

        reset();

        //Clear the vector to avoid errors. Then, copy the file contents into the vector.
        a.clear();
        a.shrink_to_fit();
        while (!inFile.eof()) {
           inFile >> x;
           a.push_back(x);
        }

        auto t5 = high_resolution_clock::now();
        MergeSort(a, 0, a.size() - 1);
        auto t6 = high_resolution_clock::now();

        duration<double, milli>MergeSortTime = t6 - t5;
        sumMergeTime += MergeSortTime.count();

        reset();

        //Clear the vector to avoid errors. Then, copy the file contents into the vector.
        a.clear();
        a.shrink_to_fit();
        while (!inFile.eof()) {
           inFile >> x;
           a.push_back(x);
        }

        cout << setw(5) << i << "\t\t" << sumBubbleTime << "\t\t"
           << setw(5) << sumInsertionTime << "\t\t"
           << setw(5) << sumMergeTime << endl;
     }

     inFile.close();
     outFile.close();
     return 0;
  }
```

Console Output

Sorting Runtime Analysis			
File Size	Bubble Sort	Insertion Sort	Merge Sort
500	0.0008	0.0002	0.0003
1000	77.294	21.4154	10.5329
1500	251.831	66.0683	24.2853
2000	554.025	154.837	44.2544
2500	1042.15	301.903	67.6642
3000	1661	516.159	93.2231
3500	2457	785.043	124.058
4000	3445.9	1175.06	167.547
4500	4686.53	1592.55	204.841
5000	6212.53	2112.84	244.192
5500	7977.05	2748.5	286.268
6000	9995.66	3438.55	331.047
6500	12460.8	4291.67	383.471
7000	15075	5250.65	435.02
7500	18180.7	6407.42	494.877
8000	21653.8	7671.78	554.119
8500	25586.2	9126.32	632.068
9000	29907.5	10839.5	699.509
9500	34887.4	12568.8	781.592
10000	40197.6	14580.3	856.207

Table 2: Comparing execution time of various sorting algorithms

A few more scripts

```
In[80]:= bubble := {{500, 0.0008}, {1000, 77.294}, {1500, 251.831}, {2000, 554.025},
           {2500, 1042.15}, {3000, 1661}, {3500, 2457}, {4000, 3445.9}, {4500, 4686.53},
           {5000, 6212.53}, {5500, 7977.05}, {6000, 9995.66}, {6500, 12460.8}, {7000, 15075},
           {7500, 18180.7}, {8000, 21653.8}, {8500, 25586.2}, {9000, 29907.5}, {9500, 34887.4},
           {10000, 40197.6}}

         nlm = NonlinearModelFit[bubble, {a * x^2 + b * x + c, a > 0}, {a, b, c}, x] // Normal
         a := ListPlot[bubble, PlotStyle → {RGBColor[1, 0, 0], PointSize[Medium]}]
         b := Plot[nlm, {x, 500, 10000}, PlotRange → Full]
         Show[b, a, AxesLabel → {"Sort Size", "Time"}, PlotLabel → "Bubble Sort O(n^2)",
          LabelStyle → Directive[Blue, Bold]]
         ClearAll["Global`*"]

Out[81]= 2002.46 - 2.08034 x + 0.000577974 x^2
```

Out[84]=

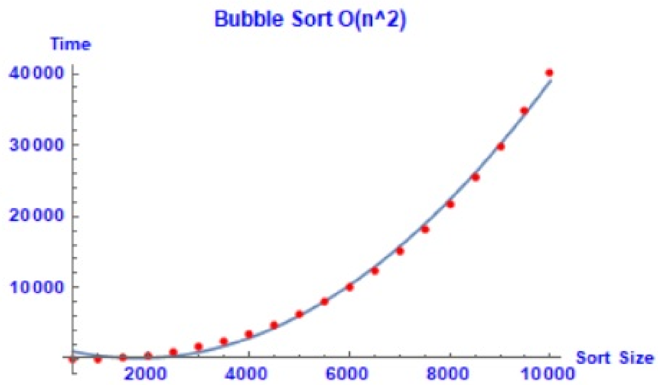

Fig 3. **Bubble Sort Complexity** $O(n^2)$

```
In[87]:= insertion := {{500, 0.0002}, {1000, 21.4154}, {1500, 66.0683}, {2000, 154.837},
    {2500, 301.903}, {3000, 516.159}, {3500, 785.043}, {4000, 1175.06}, {4500, 1592.55},
    {5000, 2112.84}, {5500, 2748.5}, {6000, 3438.55}, {6500, 4291.67}, {7000, 5250.65},
    {7500, 6407.42}, {8000, 7671.78}, {8500, 9126.32}, {9000, 10839.5}, {9500, 12568.8},
    {10000, 14580.3}}

  nlm = NonlinearModelFit[insertion, {a*x^2 + b*x + c, a > 0}, {a, b, c}, x] // Normal
  a := ListPlot[insertion, PlotStyle → {RGBColor[1, 0, 0], PointSize[Medium]}]
  b := Plot[nlm, {x, 500, 10000}, PlotRange → Full]
  Show[b, a, AxesLabel → {"Sort Size", "Time"}, PlotLabel → "Insertion Sort O(n^2)",
   LabelStyle → Directive[Blue, Bold]]
  ClearAll["Global`*"]

Out[88]= 812.225 - 0.839907 x + 0.000216857 x^2
```

Out[91]=

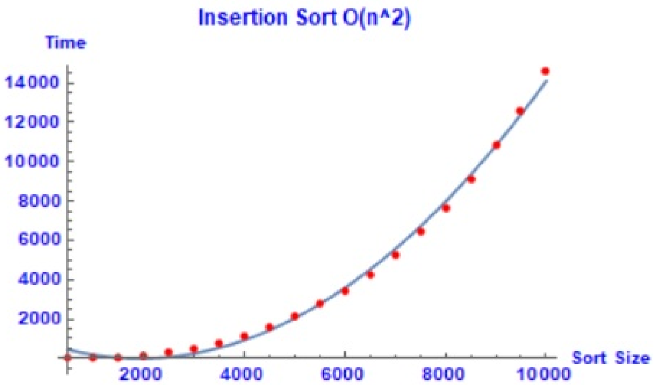

Fig 4. **Insertion Sort Complexity** $O(n^2)$

```
In[122]:= merge := {{500, 0.0004}, {1000, 11.3493}, {1500, 27.318}, {2000, 47.3594},
    {2500, 69.2014}, {3000, 97.1818}, {3500, 126.354}, {4000, 167.471}, {4500, 217.54},
    {5000, 256.305}, {5500, 300.615}, {6000, 352.666}, {6500, 402.98}, {7000, 463.746},
    {7500, 525.639}, {8000, 590.724}, {8500, 669.137}, {9000, 737.97}, {9500, 818.561},
    {10000, 906.334}}

  nlm = NonlinearModelFit[merge, {a + b*x*Log[c*x], b > 0, c > 0}, {a, b, c}, x] // Normal
  a := ListPlot[merge, PlotStyle → {RGBColor[1, 0, 0], PointSize[Medium]}]
  b := Plot[nlm, {x, 500, 10000}, PlotRange → Full]
  Show[b, a, AxesLabel → {"Sort Size", "Time"}, PlotLabel → "Merge Sort O(n log n)",
   LabelStyle → Directive[Blue, Bold]]
  ClearAll["Global`*"]

Out[123]= 82.8812 + 0.0651043 x Log[0.000340206 x]
```

Out[126]=

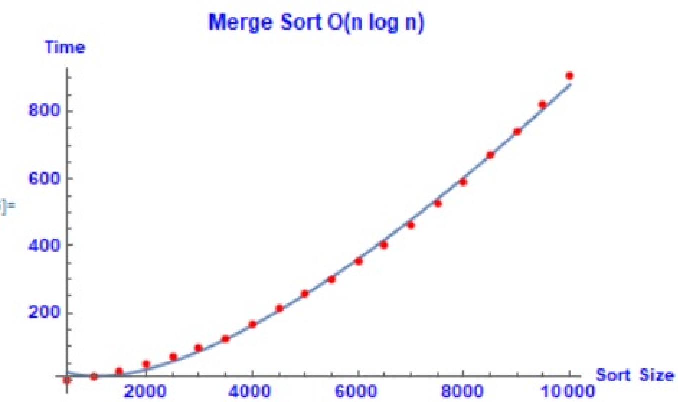

Fig 5. **Merge Sort Complexity** $O(n \log n)$

```
In[120]:= bsort[x_] := 2002.46 - 2.08034 x + 0.000577974 x^2
          isort[x_] := 812.225 - 0.839907 x + 0.000216857 x^2
          msort[x_] := 82.8812 + 0.0651043 x * Log[0.000340206 x]

          Plot[{bsort[x], isort[x], msort[x]}, {x, 0, 10000}, PlotStyle → {Black, Red, Blue},
           PlotRange → {0, 400}, PlotLegends → {"Bubble Sort", "Insertion Sort", "Merge Sort"}]
```

Out[123]=

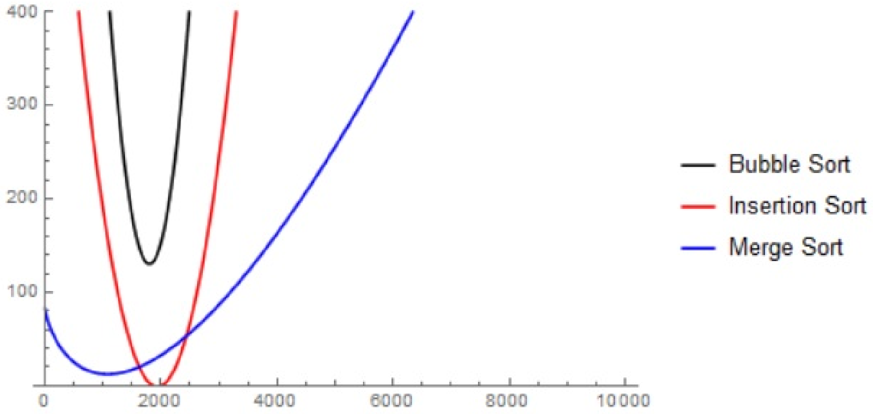

Fig 6. **Complexity comparison of sorting algorithms**

Analysis of recursive functions

- Recursive Fibonacci

The C++ code

```cpp
#include<iostream>
#include<iomanip>
#include<chrono>
using namespace std;
using namespace chrono;

unsigned long long fibonacci(int n) {
   if (n == 0 || n == 1)
      return 1;
   else
      return fibonacci(n - 1) + fibonacci(n - 2);
}

int main() {
   unsigned long long int fib;
   int n = 5;

   double sumFibTime = 0;
   cout << "Input Size\tFinal Value\tTime Taken\n";

   for (int i = 0; i < 8; i++) {
      auto t1 = high_resolution_clock::now();
      fib = fibonacci(n);
      auto t2 = high_resolution_clock::now();
      duration<double, milli>diff = t2 - t1;
      sumFibTime += diff.count();

      cout << setw(5) << n << "\t\t" << setw(9) << fib << "\t" << setw(7) << sumFibTime <<
         endl;
      n += 5;
   }
   return 0;
}
```

Console Output

Input Size	Final Value	Time Taken
5	8	0.0012
10	89	0.008
15	987	0.0732
20	10946	0.5225
25	121393	6.3224
30	1346269	65.3044
35	14930352	822.435
40	165580141	8249.97

Table 3: Recursive Fibonacci Time Analysis

Graphical Evidence

A graph of input size vs execution time, shows a exponential pattern. It is actually of the order $O(2^n)$. The following script generates the graph.

```
In[124]:= data := {{5, 0.0012}, {10, 0.008}, {15, 0.0732}, {20, 0.5225}, {25, 6.3224},
             {30, 65.3044}, {35, 822.435}, {40, 8249.97}}
          nlm = NonlinearModelFit[data, {a * 2^x + b, a > 0}, {a, b}, x] // Normal
          a := ListPlot[data, PlotStyle → {RGBColor[1, 0, 0], PointSize[Medium]}]
          b := Plot[nlm, {x, 0, 40}, PlotRange → All]
          Show[a, b, AxesLabel → {"Initial Input n", "Time in milliseconds"},
           PlotLabel → "Recursive Fibonacci O(2^n)", LabelStyle → Directive[Blue, Bold]]
          ClearAll["Global`*"]

Out[125]= 87.9946 + 7.43686 × 10^-9 × 2^x
```

Out[128]=

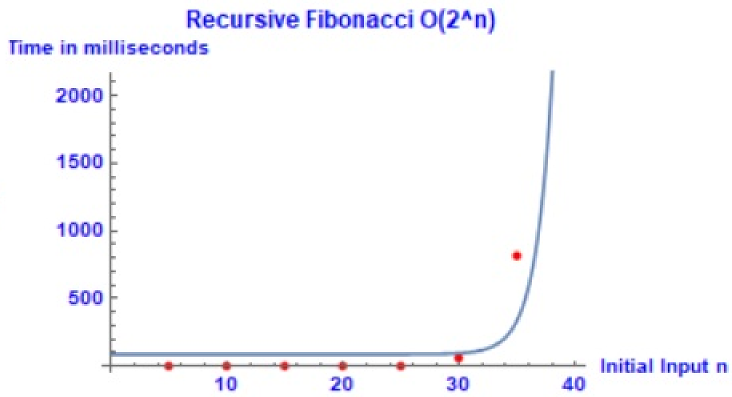

Fig 7. **Recursive Fibonacci Complexity** $O(2^n)$

- Recursive factorial

The C++ code

```
#include<iostream>
#include<iomanip>
#include<chrono>
using namespace std;
using namespace chrono;

unsigned long long factorial(int n) {
   if (n == 0 || n == 1)
      return 1;
   else
      return n * factorial(n - 1);
}

int main() {
   unsigned long long int fact;

   double sumFactTime = 0;
   cout << "Input Size\tFinal Value\tTime Taken\n";

   for (int i = 0; i < 13; i++) {
      auto t1 = high_resolution_clock::now();
      fact = factorial(i);
      auto t2 = high_resolution_clock::now();
      duration<double, milli>diff = t2 - t1;
      sumFactTime += diff.count();

      cout << setw(5) << i << "\t\t" << setw(9) << fact << "\t" << setw(7) << sumFactTime
         << endl;
   }
   return 0;
}
```

Console Output

Input Size	Final Value	Time Taken
0	1	0.0007
1	1	0.0012
2	2	0.0021
3	6	0.0029
4	24	0.0036
5	120	0.0042
6	720	0.0049
7	5040	0.0053
8	40320	0.0059
9	362880	0.0067
10	3628800	0.0078
11	39916800	0.0085
12	479001600	0.0097

Table 4: Recursive Factorial Time Analysis

Graphical Evidence

A graph of input size vs execution time, shows a linear pattern. It is actually of the order $O(n)$. The following script generates the graph.

```
In[106]:= data := {{0, 0.0007}, {1, 0.0012}, {2, 0.0021}, {3, 0.0029}, {4, 0.0036}, {5, 0.0042},
            {6, 0.0049}, {7, 0.0053}, {8, 0.0059}, {9, 0.0067}, {10, 0.0078}, {11, 0.0085},
            {12, 0.0097}}
          lm = Fit[data, {1, x}, x] // Normal
          a := ListPlot[data, PlotStyle → {RGBColor[1, 0, 0], PointSize[Medium]}]
          b := Plot[lm, {x, 0, 12}, PlotRange → All]
          Show[a, b, AxesLabel → {"Initial Input n", "Time in milliseconds"},
           PlotLabel → "Recursive Factorial O(n)", LabelStyle → Directive[Bold, Blue]]
          ClearAll["Global`*"]

Out[107]= 0.000585714 + 0.000716484 x
```

Out[110]=

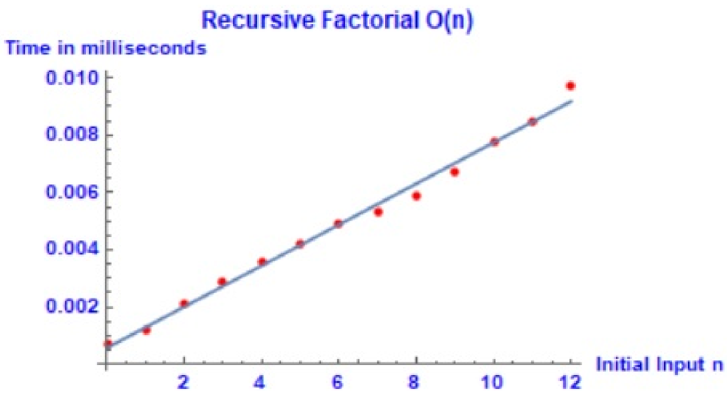

Fig 8. **Recursive Factorial Complexity** $O(n)$