Analysis and Design of Algorithms

Fernando Socualaya

April 2019

1 Warm up

As its known Merge Sort is a Divide and Conquer algorithm, it works dividing the array to be sorted into two halves until get a single array and then sorting and merging the single arrays. In order to calculate the complexity of the algorithm splitting the given array in three parts instead of two halves we have to consider that, being x the number of times that we have to split the array, and n the number of elements in the array, then $2^x = n$ or $x = \log_2 n$. Multiplying it by the size of the initial array, we get that the complexity of Merge Sort is $O(n \log_2 n)$. In that way, if we try to split in three parts instead of two halves the algorithm's complexity will be given by $n3^x = n$ or $n \log_3 n$.

2 Competitive programming

- **100 - 3n + 1**

```
#include <iostream>
#include <fstream>
using namespace std;

int tresenemasuno(long int n){
    int count = 1;

    if (n == 0){
        return n;
    }
```

```
    while(n!=1){
        if(!(n%2))
            n /= 2;
        else
            n = 3*n+1;
        ++count;
    }
    return count;
};

int getMax(int start, int end){
   int max = tresenemasuno(start);
   for(int i=start;i<=end;i++){
      if(tresenemasuno(i)>max)
         max = tresenemasuno(i);
   }
   return max;
};

int main(){
      int start=0, end=0;
   while(cin>>start>>end){
      cout<<start<<' '<<end<<' ';
      if(start>end) cout<<getMax(end,start);
      else cout<<getMax(start,end);
      cout<<'\n';
   }
   return 0;
}
```

- **458 - The Decoder**

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

int main(){
   string encoded;
   while(cin>>encoded){
      for(auto&i:encoded)
```

```
        cout<<(char)(i-7);

      cout<<endl;
   }
   return 0;
}
```

3 Simulation

- Insertion sort algorithm: Implemented at "*sortingAlgorithms.h*"

```
void insertionSort(int a[], int size){
    for(int j=1;j<size;j++){
        int key = a[j];
        int i = j-1;
        while(i>=0 and a[i]>key){
            a[i+1] = a[i];
            i--;
        }
        a[i+1] = key;
    }
```

- Merge sort algorithm: "*sortingAlgorithms.h*"

```
void Merge(int A[], int p, int q, int r){
    //Sizes for new arrays
    int n1 = q-p+1;
    int n2 = r-q;

    //let L[1 ... n1 + 1] and R[1 ... n2 + 1] be new arrays
    int L[n1+1]={};
    int R[n2+1]={};
    for(int i = 0;i<n1;i++) L[i] = A[p+i];
    for(int j = 0;j<n2;j++) R[j] = A[q+j+1];

    L[n1] = std::numeric_limits<int>::max();
    R[n2] = std::numeric_limits<int>::max();
```

```
    int i = 0;
    int j = 0;

    for(int k = p; k<=r; k++){
        if(L[i]<=R[j]){
            A[k] = L[i];
            i++;
        }
        else{
            A[k] = R[j];
            j++;
        }
    }
};

void MergeSort(int A[], int p, int r){
    if(p<r){
        int q = floor((float)(p+r)/2);
        MergeSort(A,p,q);
        MergeSort(A,q+1,r);
        Merge(A,p,q,r);
    }
};
```

- Simulations: To run the simulations was implemented a C++ program(**simulation.cpp**)to write the sizes and its respective average running time(for both algorithms) into the *simulation.csv* file. This program receive three numbers as input, the starting size, the limit size, and the number of times runned to get the average. The results obtained were plotted using the **matplotlib** library on python. As it can be seen in $Figure1$, **Merge Sort**($O(nlog(n))$) algorithm grow slower than **Insertion Sort**($O(n^2)$). However, at the beginning insertion sort grows slower than merge sort, this happens until the cut point of both curves, henceforth, the polynomial growth of insertion sort is evident in comparison to the logarithmic growth of merge sort.

Note: All algorithms are placed in the /3_Simulation folder.

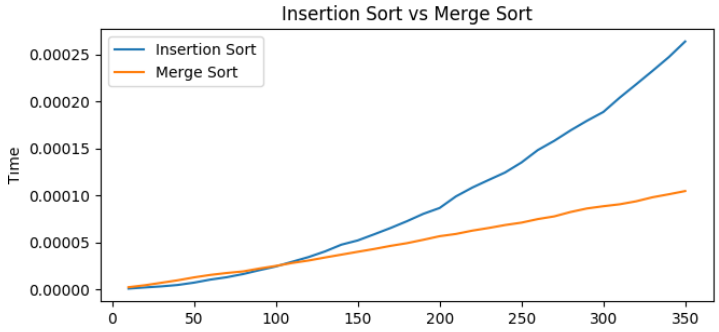

Figure 1: Algorithms Running Time Comparison

4 Research

5 Wrapping up

After comparing the complexity(Figure 2), is evident the fast growth of some ones respect others. In detail and ordered by growth:

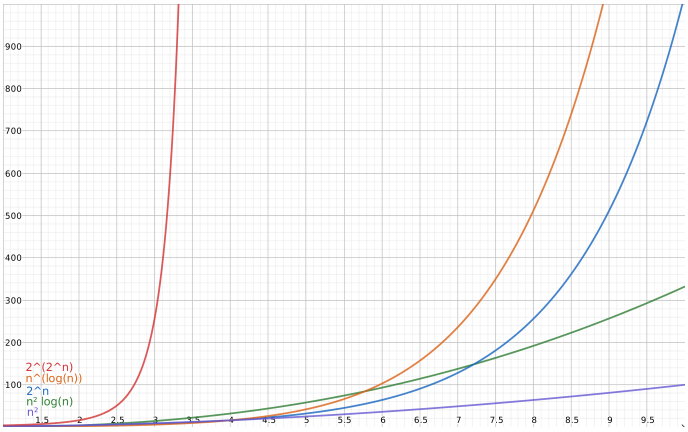

Figure 2: Algorithm complexity

1. n^2: With small values grows fast. However, it grows slower than the other ones as n grows. It ended being the slowest-growth complexity.

2. $n^2log(n)$: Started being the slowest-growth complexity. With small values growth faster than the others, but with high values it became the second fast-growth complexity.

3. 2^n: Started being the second fast-growth complexity, but after cutting $n^2log(n)$ and n^2 first, and then n^2(one more time) and $n^{log(n)}$. It ended in the third position.

4. $n^{log(n)}$: Looks like a slow-growth complexity with small values, however, as they grow, its growth turns faster. End being the second fast-growth complexity.

5. 2^{2^n}: Fast growth, it reaches high f(n) values with small n. Ended being the faster-growth complexity.