Analysis and Design of Algorithms

Jose Francisco Chavez Carreon

April 2019

1 Warm up

Lets modify the classic merge sort algorithm a little bit. What happens if instead of splitting the array in 2 parts we divide it in 3? You can assume that exists a three-way merge subroutine. What is the overall asymptotic running time of this algorithm?

Solution:

When we divide the array we just compute the middle of the subarray, this step takes constant time. Thus $D(n) = \Theta(1)$. In the Conquer step we recursively solve three subproblems, each of size $n/2$ which makes $3T(n/3)$ the running time.
The merge procedure for a n-elemet subarray will always be $\Theta(n)$, and so $C(n) = \Theta(n)$.
When we add $D(n)$ and $C(n)$ for the merge procedure we end up with the non linear function (worst case analysis).
So the worst case running time $T(n)$ of the three-way merge sort:

$$T(n) = \begin{cases} \Theta(1) & : \text{if } n = 1, \\ 3T(n/3) + \Theta(n) & : \text{if } n > 1. \end{cases}$$

2 Competitive programming

.

Figure 1: UVa judge

23111228	100 The 3n + 1 problem	Accepted	C++11	0.320	2019-04-04 19:38:37

Figure 2: UVa judge

23104508	458 The Decoder	Accepted	C++11	0.030	2019-04-03 16:18:18

Solution:

The 3n + 1 problem:

```
#include <iostream>
#include <vector>
#include <array>
#include <fstream>

int cycle(int i) {
    int cont=0;
    while (true) {
        cont++;
        if (i == 1) {
            break;
        }
        if (i % 2 != 0) {
            i = (3 * i) + 1;
        } else {
            i = i / 2;
        }
    }
    return cont;
}

int main(){


    std::array<int,3> arr{};
    int a,b;
    int aloj;
    while(std::cin>>a>>b){

        bool switchs=false;
        if(a>b && a>=0){
            int temp=a;
            a=b;
            b=temp;
```

```
            switchs=true;
        }
        int container=0;
        for(int i=a;i<=b;i++){
            aloj=cycle(i);
            if(aloj>container){
                container=aloj;
            }
        }
        if(switchs){
            arr[0]=b;
            arr[1]=a;
        }else{
            arr[0]=a;
            arr[1]=b;
        }
        arr[2]=container;
        std::cout<<arr[0]<<" "<<arr[1]<<" "<<arr[2]<<'\n';
        //lista.push_back(arr);
    }




    return 0;

}
```

Solution:

The decoder problem:

```
#include <iostream>
#include <vector>

std::string decode(std::string linea){
    std::string new_string;
    for(auto &i : linea){
        new_string+=(i-7);
    }
    return new_string;
}

int main(){
    std::string a;
```

```
    while(std::cin>>a){
        std::cout<<decode(a)<<'\n';
    }
    return 0;
}
```

3 Simulation

Write a program to find the minimum input size for which the merge sort algorithm always beats the insertion sort.

- Implement the insertion sort algorithm

```
    void inSort(int *arr,int size){
    for(int j=1;j<size;j++){
        int key=arr[j];
        int i = j - 1;
        while (i>-1 && arr[i]>key){
            arr[i+1]=arr[i];
            i=i-1;
        }
        arr[i+1]=key;
    }
}
```

- Implement the merge sort algorithm

```
    #include <limits>

void merge(int A[],int p,int q,int r){
    int INF = std::numeric_limits<int>::max();
    int n1= q-p+1;
    int n2= r-q;
    int *L,*R;
    L=new int[n1+1];
    R=new int[n2+1];
    for(int i=0;i<n1;i++){
        L[i]=A[p+i];
    }
    for(int i=0;i<n2;i++){
        R[i]=A[(q+1)+i];
```

```
    }
    L[n1]=INF;
    R[n2]=INF;
    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++;
        }
    }
    delete L;
    delete R;
}

void mergesort(int A[],int p,int r){
    if(p<r){
        int q=(p+r)/2;
        mergesort(A,p,q);
        mergesort(A,q+1,r);
        merge(A,p,q,r);
    }
}
```

- 3erd algorithm to compare: **Quicksort**

```
    void Swap(int arr[],int index1, int index2){

    int exchange_var;
    exchange_var=arr[index2];
    arr[index2]=arr[index1];
    arr[index1]=exchange_var;


}

int PartitionArray(int arr[],int p,int r){

    int x = arr[r];
    int i = p-1;
```

```
    for(int j=p;j<=r-1;j++)
    {
        if(arr[j]<=x){

            i+=1;
            Swap(arr,i,j);

        }
    }

    Swap(arr,i+1,r);

    return i + 1;
}
    void Quicksort(int arr[],int p,int r){

    if(p<r){

        int x = PartitionArray(arr,p,r);

        Quicksort(arr,p,x-1);
        Quicksort(arr,x+1,r);

    }
}
```

From now on , Merge Sort will have the orange color, Insertion Sort will have the green color and Quick sort will be the blue color.

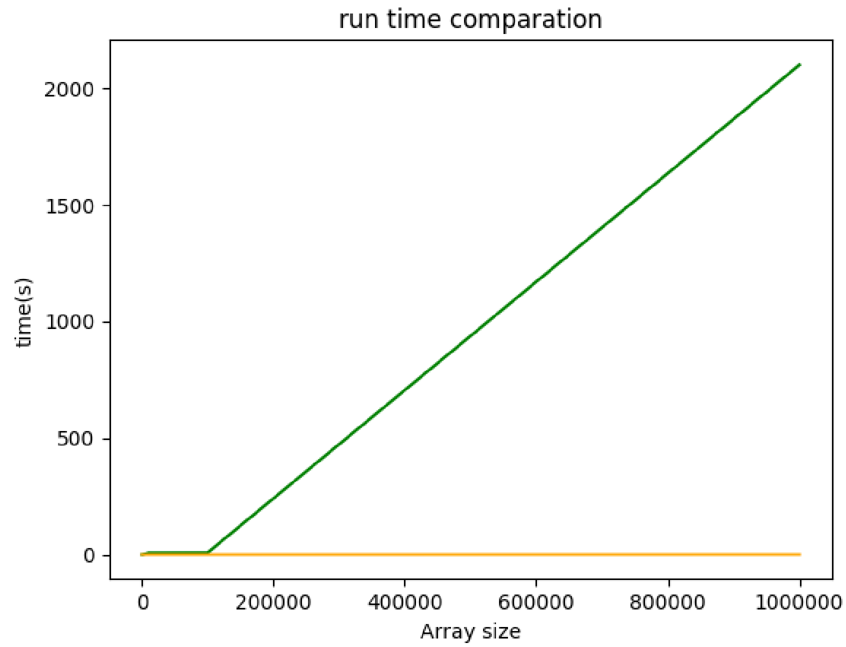

Figure 3: Here we are seeing an astonishing victory of the merge sort

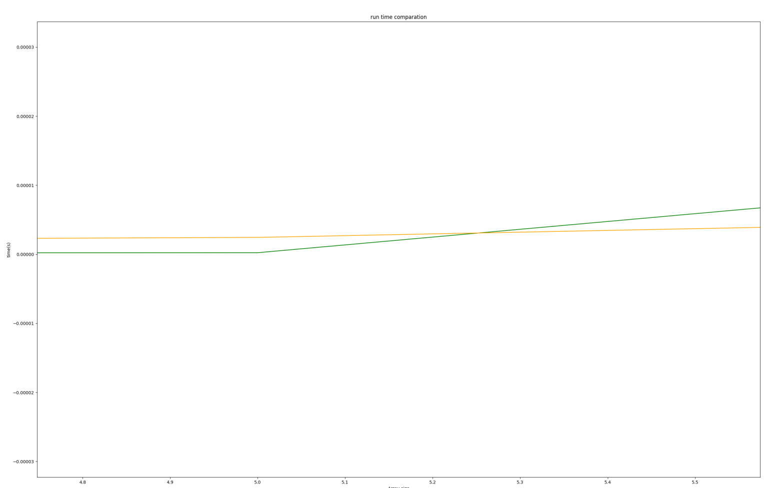

Figure 4: Now we see that apparently insertion sort is better than the merge sort in really small arrays, here Insertion Sort wins in arrays lower than 6 number

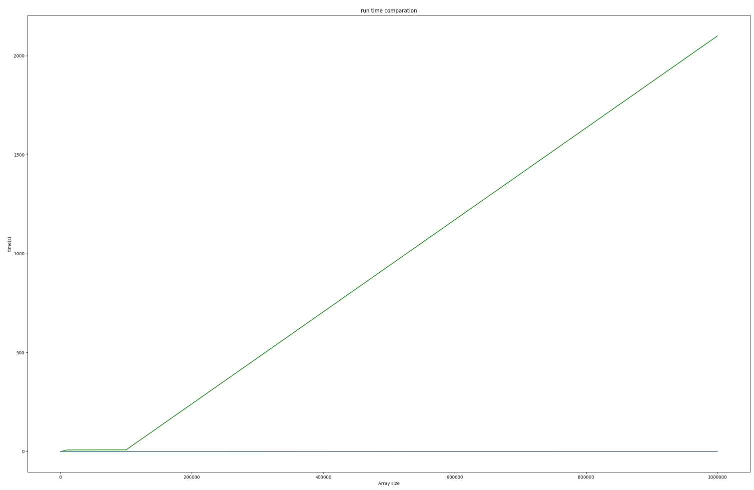

Figure 5: QuickSort and MergeSort have similar times, that is why QuickSort line is above MergeSort line

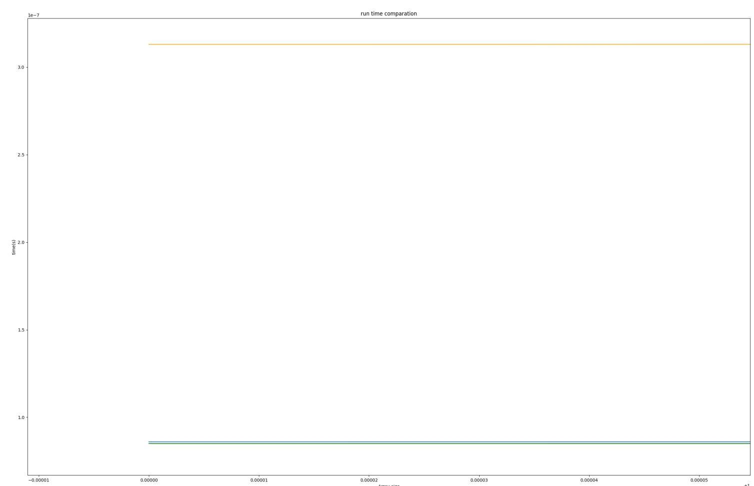

Figure 6: Here is a closer look to the algorithms

4 Research

Everybody at this point remembers the quadratic "grade school" algorithm to multiply 2 numbers of k_1 and k_2 digits respectively.

Your assignment now is to compare the number of operations performed by the quadratic grade school algorithm and Karatsuba multiplication.

- Define Karatsuba multiplication
  Karatsuba multiplication is an algorithm discovered by Anatoly Karatsuba, it is a fastest way to multiplicate numbers. It works by decomposing the numbers in equal length sets to operate them, this step transforms a big multiplication in two smaller multiplications. First, we start with the first equation that will decompose our numbers in equal length sets.
  $10^{n/2}a + b$. Where a and b are positive integers and n is the length of the number.
  We will apply the same equation to the other number, but we will end with c and d instead.
  So *number1* will be equal to $10^{n/2}a + b$ and *number2* will be $10^{n/2}c + d$
  We will multiply both expressions and end with $10^{n}ac + 10^{n/2}(ad + bc) + bd$
  We end up with just 3 operations, but this 3 can end up beeing 2, lets see.
  we will make this simple add an multiply equation $(a + b)(c + d) = ac + ad + bc + bd$.
  We notice that we already have ac and bd, so we will only have substract these therms to the equation result and the we will have the full $(ad + bc)$ expression.
  So this is how karatsuba algorithm simplifies the common way to multiply numbers.

- Implement grade school multiplication

```
std::string SumaColumna(std::vector<std::string> numeros){
    unsigned long size=0;
    for(auto &i: numeros){
        if(i.size()>size){
            size=i.size();
```

```
        }
    }

    int sum_holder=0;
    int sum=0;
    std::string numero;
    std::string extra;
    for(int i=0;i<size;i++){
        for(auto &j : numeros){
            if(i<j.size()) {
                extra=j[(j.size()-1)-i];
                sum += stoi(extra);
            }
        }
        sum+=sum_holder;
        if(i==size-1){
            numero.insert(0,std::to_string(sum));
        }
        else{

            sum_holder= (sum/10);
            sum=sum%10;
            numero.insert(0,std::to_string(sum));
            sum=0;

        }
    }
    return numero;
}

std::string multiply(std::string a, std::string b){
    std::vector<std::string> numeros;
    std::string string_holder;
    int num_holder=0;
    int  sum_holder=0;
    int mult=0;
    if(a>=b){
        for(int i=b.size()-1;i>=0;i--){

            for(int j=a.size()-1;j>=0;j--){
                std::string d="";
                std::string s="";
                d+=b[i];
                s+=a[j];
                mult=(stoi(d)* stoi(s))+sum_holder;
```

```
                num_holder=mult%10;
                sum_holder=mult/10;
                if(j==0){
                    string_holder.insert(0,std::to_string(mult));

                }else{
                string_holder.insert(0,std::to_string(num_holder));
                }
                d="";
                s="";
            }
            for(int k=1;k<b.size()-i;k++){
                string_holder+="0";
            }
            numeros.push_back(string_holder);
            string_holder="";
            sum_holder=0;
        }
    }
    else{
        auto temp=a;
        a=b;
        b=temp;

        for(int i=b.size()-1;i>=0;i--){

            for(int j=a.size()-1;j>=0;j--){
                std::string d="";
                std::string s="";
                d+=b[i];
                s+=a[j];
                mult=(stoi(d)* stoi(s))+sum_holder;
                num_holder=mult%10;
                sum_holder=mult/10;
                if(j==0){
                    string_holder.insert(0,std::to_string(mult));

                }else{
                    string_holder.insert(0,std::to_string(num_holder));
                }
                d="";
                s="";

            }
            for(int k=1;k<b.size()-i;k++){
```

```
                string_holder+="0";
            }
            numeros.push_back(string_holder);
            string_holder="";
            sum_holder=0;
            num_holder=0;
        }
    }
    std::string final=SumaColumna(numeros);
    return final;
}
```

- Implement Karatsuba multiplication

```
    std::string karatsuba(std::string number1,std::string number2){
    /*
     * separating numbers in two sets
     * */

    if(number1.size()==1 && number2.size()==1){
        return std::to_string(std::stoi(number1)*std::stoi(number2));
    }
    std::string aa,bb,cc,dd;
    int half;
    if(number1.size()%2==0) {
        half = number1.size() / 2;
    }else{
        half = (number1.size() / 2)+1;
    }
    bool switchs=false;
    int cont=0;
    for(auto &i : number1){
        if(cont==half){
            switchs=true;
            cont++;
        }
        if(!switchs){
            aa+=i;
            cont++;
        }else{
            bb+=i;
        }
    }
    int half1;
    if(number2.size()%2==0) {
        half1 = number2.size() / 2;
```

```
    }else{
        half1 = (number2.size() / 2)+1;
    }
    bool switchs1=false;
    int cont1=0;
    for(auto &i : number2){
        if(cont1==half1){
            switchs1=true;
            cont1++;
        }
        if(!switchs1){
            cc+=i;
            cont1++;
        }else{
            dd+=i;
        }
    }
/*
 * multiplication process
 * */

    std::string ac,bd;
    if(number1.size()>2){
        ac=karatsuba(aa,cc);
        bd=karatsuba(bb,dd);
    }else{
        ac=std::to_string(stoi(aa)*stoi(cc));
        bd=std::to_string(stoi(bb)*stoi(dd));
    }

    unsigned long int a,b,c,d;

    a=stoi(aa);
    b=stoi(bb);
    c=stoi(cc);
    d=stoi(dd);

    unsigned long int mid_term=(a+b)*(c+d);

    unsigned long int nac=stoi(ac);
    unsigned long int nbd=stoi(bd);

    unsigned long int  first_term;
    unsigned long int second_term;
```

```
    if(number1.size()%2==0){
        first_term=(pow(10,number1.size())*nac);
        second_term=(pow(10,(number2.size()/2))*(mid_term-(nac+nbd)));
    }else{
        first_term=(pow(10,number1.size()-1)*nac);
        second_term=(pow(10,((int)(number2.size()/2)))*(mid_term-(nac+nbd)));
    }

    unsigned long int ecuation=first_term+second_term+nbd;
    return std::to_string(ecuation);
}
```

- Compare Karatsuba algorithm against grade school multiplication

  Karatsuba algorithm reduces the multiplication of two numbers to a $n^{1.58}$ single digits operation. The classic algorithm (grade school algorithm) requires n^2 single digit operations. So, the Karatsuba algorithm is less cost than the classic algorithm.

- Use any of your implemented algorithms to multiply $a * b$ where:

  a: 3141592653589793238462643383279502884197169399375105820974944592

  b: 2718281828459045235360287471352662497757247093699959574966967627

  Using grade school algorithm:
  $a{*}b =$ 8539734222673567065463550869546574495034888535765114961879601127067743044893204848617875072216249073013374895871952806582723184

5 Wrapping up

Arrange the following functions in increasing order of growth rate with $g(n)$ following $f(n)$ if $f(n) = \mathcal{O}(g(n))$

1. $n^2 log(n)$
2. 2^n
3. 2^{2^n}
4. $n^{log(n)}$
5. n^2

Solution:
So the new order will be:

1. 2^{2^n}
2. 2^n
3. $n^{log(n)}$
4. $n^2 log(n)$
5. n^2

We can notice this with just looking at the chart

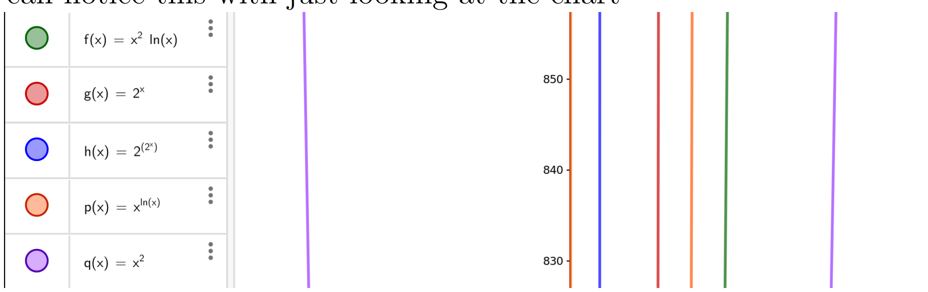

Figure 7: Here we have all the equations

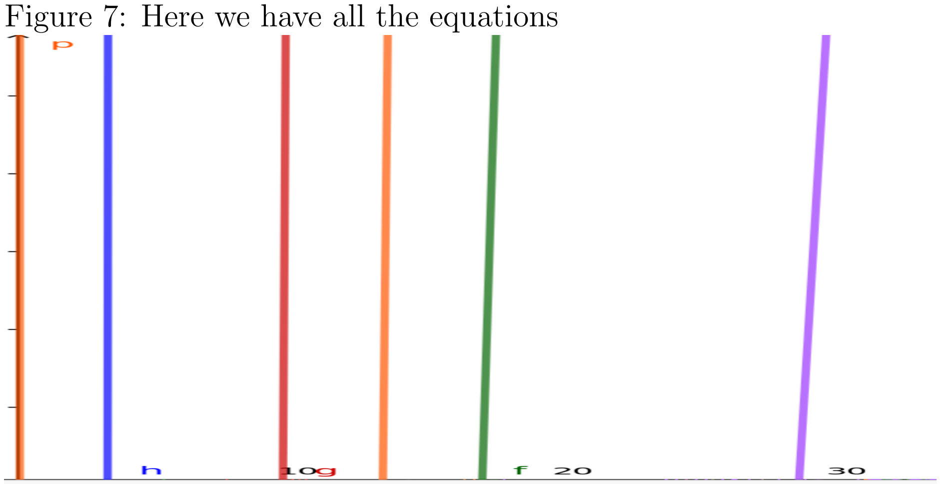

Figure 8: This is a closer look