Sesión 3: Divide y vencerás I

Marzo 31, 2020

1. Diseño de algoritmos

1. Analiza la siguiente regla para calcular el máximo común divisor

$$\text{mcd}(a,b) = \begin{cases} 2\text{mcd}(a/2, b/2) & \text{si } a, b \text{ son pares} \\ \text{mcd}(a, b/2) & \text{si } a \text{ es impar, } b \text{ es par} \\ \text{mcd}((a-b)/2, b) & \text{si } a, b \text{ son impares} \end{cases}$$

a) ¿Es correcta? es decir, al utilizarla ¿es posible calcular correctamente el mcd? Justifica tu respuestas

b) Basándote en ella, escribe el pseudocódigo de un algoritmo que utilice la estrategia de divide y vencerás para calcula el mcd.

c) Calcula la complejidad de dicho algoritmo

a)

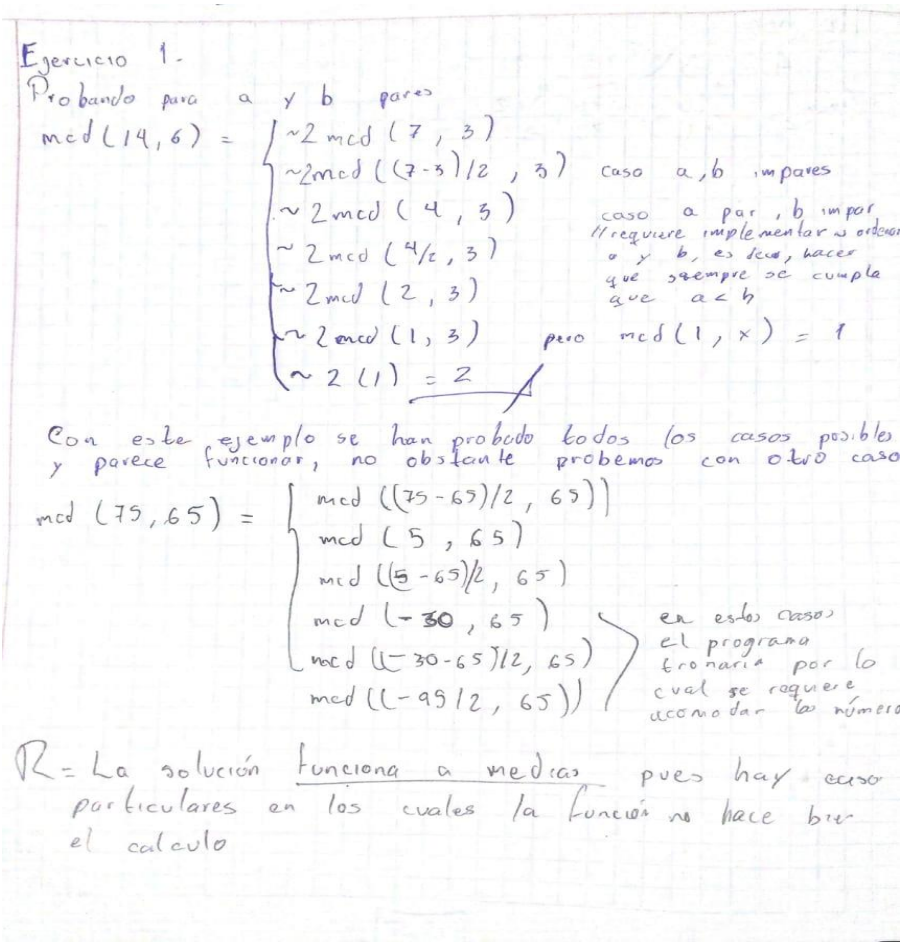
Ejercicio 1.
Probando para a y b pares
mcd(14,6) = { ~2 mcd(7,3); ~2mcd((7-3)/2, 3); ~2 mcd(4,3); ~2 mcd(4/2,3); ~2 mcd(2,3); ~2 mcd(1,3); ~2(1) = 2 }
caso a, b impares
caso a par, b impar
//requiere implementar u orden a y b, es fácil, hacer que siempre se cumpla que a < b
pero mcd(1, x) = 1
Con este ejemplo se han probado todos los casos posibles y parece funcionar, no obstante probemos con otro caso
mcd(75,65) = { mcd((75-65)/2, 65); mcd(5, 65); mcd((5-65)/2, 65); mcd(-30, 65); mcd((-30-65)/2, 65); mcd((-95)/2, 65) }
en estos casos el programa tronaría por lo cual se requiere acomodar los números
R = La solución funciona a medias pues hay casos particulares en los cuales la función no hace bien el calculo

Ilustración 1 Ejemplo ejercicio 1

```c
int mcd(int a, int b){
    if (a == b)
        return a;
    else if (a % 2 == 0 && b % 2 == 0)      //Ambos pares
        return 2 * mcd(a/2, b/2);
    else if (a % 2 == 1 && b % 2 == 0)      //a impar, b par
        return mcd(a, b/2);
    else if (a %  2 == 0 && b %2 == 1)      //a par, b impar
        return mcd(a/2, b);
    else if (a % 2 == 1 && b % 2 == 1)      //Ambos impares
        if(a>b)
            return mcd((a-b)/2, b);     //subcaso a > b
        else
            return mcd((b-a)/2, a);     //subcaso a < b
}
```

Ilustración 2 Primera versión mcd

Tras realizar algunos pequeños ajustes en el algoritmo original, verificamos con una prueba de escritorio que el algoritmo funciona correctamente.

```
D:\ESCOM\A_ALGORITMOS\P3\Ejercicio1>ejer1a.exe

Ingrese el primer numero: 75

Ingrese el segundo numero: 65

El Maximo Comun Divisor es: 5
```

Ilustración 3 Prueba de funcionamiento ejercicio 1a

b) Si modificamos el algoritmo del inciso a observaremos que el problema puede ser dividido en 3 casos básicos, siendo el primero de ellos a = b, el segundo a > b y finalmente a < b. Al observar el primer algoritmo observamos que bastaría con hacer una resta entre ambos factores, lo cual

```c
int gcd(int m, int n) {
   if(m == n)
      return m;
   else if (m > n)
      return gcd(m-n, n);
   else
      return gcd(m, n-m);
}
```

Ilustración 4 Segunda versión gcd (Euclidean Algorithm)

Verificamos que la nueva modificación de nuestro algoritmo funcione correctamente mediante una prueba de escritorio.

```
D:\ESCOM\A_ALGORITMOS\P3\Ejercicio1>ejer1b.exe

Ingrese el primer numero: 75

Ingrese el segundo numero: 65

El Maximo Comun Divisor es: 5
```

Ilustración 5 Prueba de escritorio ejercicio 1b

c) Vamos a estimar la complejidad del tiempo de este algoritmo (basado en n = a + b). El número de pasos puede ser lineal, por ejemplo, mcd (x, 1), por lo que la complejidad es $O(n)$. Esta es la complejidad del peor de los casos, porque el valor x + y disminuye con cada paso.

2. Diseña un algoritmo que dado un arreglo con los coeficientes de un polinomio $p(x) = a_0 + a_1x + a_2x^2 + \cdots + a_nx^n$ y un valor a, devuelva la evaluación del polinomio en a, es decir, el valor de $p(a)$. Observa que el algoritmo más trivial tendrá complejidad $O(n^2)$, diseña un algoritmo que utilice la estrategia de divide y vencerás. Calcula la complejidad de tu algoritmo.

Aunque la solución de un polinomio para un valor específico de x es una tarea sencilla el algoritmo reduce la cantidad de operaciones necesarias para llegar al resultado lo que la convierte en una técnica más eficiente y deseable a la hora de programarla.

Llamando a el grado del polinomio g una resolución por sustituciones requiere hasta $(g^2 + g)/2$ multiplicaciones y g sumas mientras que el algoritmo de Horner solo requerirá g sumas y g multiplicaciones.

A continuación, se muestra un ejemplo de la regla de Horner.

Sea el polinomio de grado 3:

$$4x^3 + 3x^2 + 2x + 1; \quad x = 4$$

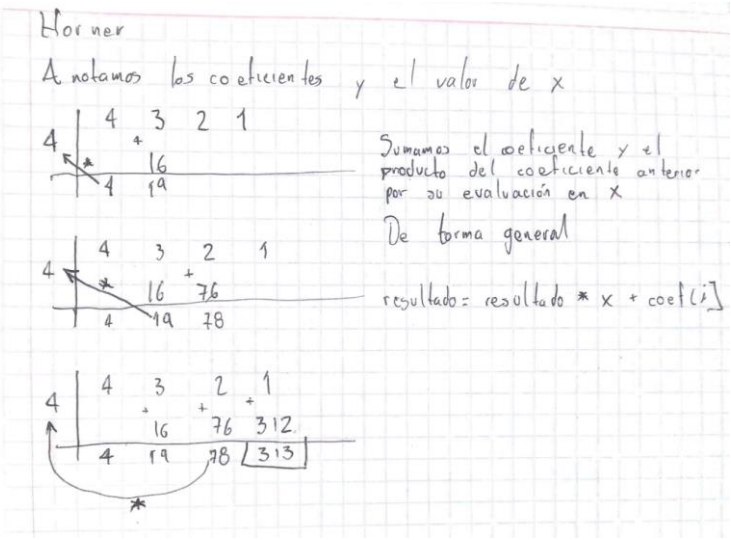

Ilustración 6 Ejemplo método de Horner

```c
int horner(int *coef, int x){
    int i;
    int resultado = 0;
    for(i=0; i < sizeof(coef); i++)
        resultado = resultado * x + coef[i];
    return resultado;
}
```

Ilustración 7 Algoritmo de Horner

```
D:\ESCOM\A_ALGORITMOS\P3\Ejercicio2>horner.exe

Introduzca el valor de X a evaluar: 4

El resultado de la evaluacion es: 313
```

Ilustración 8 Pruebe de escritorio ejercicio 2

Complejidad del algoritmo

Dado que nuestro algoritmo solamente recorre una vez el arreglo de coeficientes con un ciclo for, y dado que el resto de las operaciones en el algoritmo son asignaciones, se puede decir que el algoritmo tiene una complejidad de $\boldsymbol{O(n)}$.

3. Dado un arreglo A de n enteros distintos, se quiere contar el número de pares de índices (i, j) tales que $i < j$ y $A[i] > 2A[j]$.

   - Diseña un algoritmo por fuerza bruta para solucionar el problema anterior. Realiza el análisis correspondiente para determinar la complejidad de este algoritmo.
   - Diseña una segunda versión para solucionar el problema, cuya complejidad sea $O(n \log n)$.

   En ambos casos, muestra con un ejemplo el funcionamiento de tus algoritmos.

```c
int num(int *a, int n){
    int i, j;
    int count = 0;
    for (i = 0; i < n; i++)
    {
        for (j = i+1; j < n; j++)
        {
            if (a[i] > 2*a[j]){
                count++;
                cout<<"("<<a[i]<<" , "<<a[j]<<")"<<endl;
            }
        }
    }
    cout<<"\n\nEl numero de pares es: "<<count<<endl;
    return count;
}
```

Ilustración 9 Inversiòn por fuerza bruta

```
D:\ESCOM\A_ALGORITMOS\P3\Ejercicio3>ejer3.exe
Numero de elementos en el arreglo:
9
|3|10|11|4|12|9|7|8|6
(10 , 4)
(11 , 4)


El numero de pares es: 2
```

Ilustración 10 Prueba escritorio F. bruta

a) En este caso es un poco obvio que el algoritmo tendrá que recorrer el arreglo a través de dos ciclos for. El primero de ellos se encarga de barrer el arreglo en la posición i, mientras que el segundo evalúa el elemento en a[i] con respecto a los demás elementos del arreglo, por lo cual es fácil intuir que la complejidad de nuestro algoritmo es de $\boldsymbol{O(n)}$.

```
int mergeSort(int vec[], int s, int e)
    {
        if(s>=e) return 0;
        int mid=(s+e)/2, Inv1, Inv2, Inv3;
        int aux[100];
        Inv1=mergeSort(vec,s,mid);
        Inv2=mergeSort(vec,mid+1,e);
        Inv3=mergeInv(vec,aux,s,e);
        return Inv1+Inv2+Inv3;
    }

int mergeInv(int vec[], int aux[], int s, int e)
    {
        int mid=(s+e)/2, i=s, j=mid+1, k=s, Inv=0;

        while(i<=mid && j<=e)
            {
                if(vec[i] < vec[j])
                    {
                        aux[k]=vec[i];
                        k++;i++;
                    }
                else
                    {
                        aux[k]=vec[j];
                        if(vec[i] > vec[j]*2) Inv=Inv+(mid-i+1);
                        k++;j++;
                    }
            }
        while(i<=mid) aux[k++]=vec[i++];
        while(j<=e) aux[k++]=vec[j++];
        for(i=s;i<=e;i++) vec[i]=aux[i];
        return Inv;

    }
```

Ilustración 11 Inversión con Merge

```
D:\ESCOM\A_ALGORITMOS\P3\Ejercicio3>ejer3b.exe

El numero de inversiones es: 2
```

Ilustración 12 Prueba de escritorio inversion merge

b) Al usar Mergesort observamos que el arreglo está siendo ordenado de algún modo, al mismo tiempo que va haciendo un conteo de las inversiones que existen en el arreglo. Dado que se hace uso de la función Mergesort misma que ha sido previamente trabajada como algoritmo de ordenamiento, es fácil saber que la complejidad del algoritmo será $\boldsymbol{O}(\boldsymbol{nlogn})$.