Trabajo Práctico N°2

Complejidad Computacional

Integrantes:

* Perez Lautaro
* Quinteiro Lucas
* Mediotte Facundo
* De Rito Micaela

Profesores:

* Aubin Verónica
* Blautzik Leonardo
* Ponce De León Lucas
* Videla Lucas

Análisis de Complejidad Computacional

2.1 -

Multiplicaciones Sucesivas:

public double evaluarMSucesivas(double x) {

 φ(1)

double resultado = 0;

 φ(1)

double multSuc = 1;

for(int i=0;i<coeficientes.length;i++) {

multSuc = 1;

Al tener dos for anidados, se multiplican las complejidades

φ(n). φ(n) = φ(n2)

for(int j = 0;j<grado-i;j++) {

multSuc\*=x;

 φ(n)

 φ(n)

}

resultado+= coeficientes[i]\*multSuc;

}

return resultado;

}

TOTAL:  φ(1) +  φ(1) +  φ(n2) = φ(n2)

Recursiva:

public double evaluarRecursiva(double x) {

 φ(1)

double potencia;

 φ(1)

double suma = 0;

for(int i = 0; i<coeficientes.length;i++) {

potencia = potencia(x,grado-i);

 φ(n)

suma+=coeficientes[i]\*potencia;

}

return suma;

}

TOTAL:  φ(1) +  φ(1) +  φ(n) = φ(n)

Recursiva Par:

public double evaluarRecursivaPar(double x) {

 φ(1)

double potencia = 1;

 φ(1)

double suma = 0;

 φ(1)

int n = grado;

for(int i = 0; i<coeficientes.length;i++) {

if( n % 2 == 0)

potencia = potencia(x\*x,n/2);

if( n % 2 != 0)

 φ(n)

potencia = x\*potencia(x,n-1);

suma+=coeficientes[i]\*potencia;

n--;

}

return suma;

}

TOTAL:  φ(1) +  φ(1) + φ(1) +  φ(n) = φ(n)

Programación Dinámica:

public double evaluarProgDinamica(double x) {

 φ(1)

double resultado = coeficientes[grado];

 φ(1)

double potencia = 1;

for(int i = grado; i > 0; i--) {

potencia\*=x;

 φ(n)

resultado+= potencia\*coeficientes[i-1];

}

return resultado;

}

TOTAL:  φ(1) +  φ(1) +  φ(n) = φ(n)

Mejorada:

public double evaluarMejorada(double x) {

 φ(1)

double suma = coeficientes[grado] + coeficientes[grado-1]\*x;

 φ(1)

int potactual = 1;

 φ(1)

int potant = 0;

 φ(1)

int gradoevaluado = 0;

for(int i=grado-1; i>=grado/2; i--) {

gradoevaluado = grado - i;

potactual\*=x;

if(i- gradoevaluado >=0)

 φ(n)

suma+=potactual \* potactual \* coeficientes[i - gradoevaluado];

suma+= potactual \* potant \* coeficientes[i- gradoevaluado + 1];

potant = potactual;

}

return suma;

}

TOTAL:  φ(1) + φ(1) + φ(1) + φ(1) +  φ(n) = φ(n)

Math.Pow:

public double evaluarPow(double x) {

 φ(1)

double resultado = 0;

for(int i=0;i<coeficientes.length;i++) {

resultado+= coeficientes[i]\*Math.pow(x, grado-i);

 φ(n)

}

return resultado;

}

TOTAL:  φ(1) +  φ(n) = φ(n)

Horner:

public double evaluarHorner(double x) {

 φ(1)

double suma = coeficientes[0];

for(int i = 1; i<=grado ; i++) {

suma\*=x;

 φ(n)

suma+=coeficientes[i];

}

return suma;

}

TOTAL:  φ(1) +  φ(n) = φ(n)

2.2 –

Binomio de newton usando métodos no recursivos:

public long desarrolloCompletoDelBinomio(int x){

long resultado =0;

 φ(n2)

long[] coeficientes = obtenerCoeficientes();

for(int k = 0; k <= n; k ++){

 φ(n)

resultado +=coeficientes[k] \* Math.*pow*(this.a\*x, this.n -k) \*

Math.*pow*(this.b, k);

}

return resultado;

}

TOTAL: φ(n2) + φ(n) = φ(n2)

Binomio de newton usando métodos recursivos:

public long desarrolloCompletoDelBinomioRecursivo(int x){

long resultado = 0;

 φ(n2)

long[] coeficientes = obtenerCoeficientesRecursivo();

for(int k = 0; k <= n; k ++){

 φ(n)

resultado += coeficientes[k] \* Math.*pow*(this.a\*x, this.n -k)

\* Math.*pow*(this.b, k);

}

return resultado;

}

TOTAL: φ(n2) + φ(n) = φ(n2)

Binomio de newton usando programación dinámica con memorización:

public long desarrolloCompletoDelBinomioProgDinamicaMemorizando(int x) {

long resultado = 0;

llenarTrianguloPascal(this.n);

 φ(n)

for(int k = 0; k <= n; k ++){

 φ(n)

resultado += *pascal*[n][k] \* Math.*pow*(this.a\*x, this.n -k)

\* Math.*pow*(this.b, k);

}

return resultado;

}

TOTAL: φ(n) + φ(n) = φ(n)