Computacíon y Algoritmos

Alonso Ramírez Manzanares
Universidad de Guanajuato
Enero – Julio 2012

Eugenio Daniel Flores Alatorre

Objetivos del curso

- Entender los conceptos de algoritmo y estructura de datos
- Entender la importancia de un buen diseño de algoritmos
- Al final del curso que sean capaces de resolver problemas e implementarlos en una computadora utilizando lenguaje de alto nivel. (C++)
- Que sepan entender y analizar el desempeño de un algoritmo y determinar las estructuras de datos adecuadas para su implementación.
- Motivarlos que el área de análisis de algoritmos es muy interesante tanto para matemáticos como para computólogos (hay una intersección enorme).

Pre-requisitos

Conocimientos de programación en C, C++.

Evaluación

Exámenes (2 o 3)	35%
Proyecto (en 2 entregas)	30%
Tareas	30%
Exámenes rápidos	5%

Temario

1) Introducción al análisis y diseño de algoritmos
2) Conceptos básicos para analizar algoritmos: notación asintótica.
3) Estructuras de datos básicas: tipos de datos, arreglos, cadenas, listas ligadas, pilas, colas.
4) Recursión, recursión con memoria.
5) Algoritmos fundamentales: búsqueda y ordenamiento.
6) Algoritmos de grafos y árboles.
7) Estrategias de implementación y diseño de algoritmos: fuerza bruta, algoritmos glotones, dividir para vencer, programación dinámica.

Bibliografía

- R. Sedgewick. *Algorithms in C++.*
- B. Preiss. *Data structures and Algorithms with Object Oriented Design Patterns in C++, Java.*
- C. Cormen, C. Leiserson, R. Rivest y C. Stein. *Introduction to algorithms.*
- J. Kleinberg y E. Tardos. *Algorithms Design.*
- D. Knut. *The art of computer programming.*

Modo de programación

Durante el curso usamos Code::Blocks como ambiente de programación, que tiene tanto un editor, un debugger y un compilador (MinGW).

MinGW es compatible en multiplataformas bajo las normas ANSI.

Qué es un algoritmo

Un algoritmo es un método bien definido para resolver problemas. Es un procedimiento computacional que toma un conjunto de valores como entrada y produce un valor como salida.

Es un concepto similar a receta, proceso, método, técnica, procedimiento o rutina. Es un conjunto finito de reglas que da una secuencia de operaciones para resolver un problema específico. Debe ser finito, bien definido, tener entradas, salidas y ser factible.

Un buen algoritmo es eficiente en el uso de recursos como tiempo de ejecución o memoria de máquina, además de su adaptabilidad a diferentes computadoras, la modularidad, que sea correcto, que sea mantenible, su funcionalidad, robustez, seguridad, amigable al usuario, simple, extensible y confiable.

Un buen algoritmo reduce el tiempo de ejecución mejor que una máquina con mayor velocidad.

Comparando algoritmos

Una manera de comparar algoritmos es mediante la velocidad. Si tres algoritmos resuelven correctamente un problema, entonces el mejor será el más veloz. La velocidad es, en muchos casos, símbolo de un mejor algoritmo.

Medimos la complejidad de un algoritmo en términos de la cantidad de operaciones necesarias para llevarlo a cabo. La siguiente tabla es de utilidad, con las complejidades explicadas y ordenadas de menor a mayor:

1	Constante	Es independiente del tamaño del problema
logN	Logarítmico	Ocurre generalmente en problemas grandes que se dividen en una serie de problemas más pequeños.
N	Lineal	Cuando se hace un procesamiento por cada elemento que entra.
NlogN	N-logarítmico	Problemas donde cada iteración se divide en subproblemas donde el procesamiento es lineal.
N^2	Cuadrático	Caso de algoritmos que tienen ciclos anidados.
N^3	Cúbico	Tres ciclos anidados.

Polinomial		Se dice que un algoritmo de complejidad polinomial es realizable. Sin embargo, esto deja de ser cierto desde exponentes no muy grandes y cantidad de entradas moderadas.
2^N	Exponencial	Estos algoritmos son generalmente ingenuos y a partir de ciertos tamaños pequeños se vuelven irrealizables.

Tarea 0

Investigar cómo se usa el debugger de Code::Blocks y cada una de las funciones:

> continue, next line, next instruction, step into, step out, toggle break point, remove all break points, run to cursor, debugging windows, edit watches.

El debugger es una manera distinta de compilar el proyecto. *Watches*, como su nombre bien lo indica, nos permite tener en la mira cualquier cosa que nosotros queramos; por default tiene las variables locales de una función –que, por default serán las de main a menos que entremos en alguna otra función- y los argumentos de la función –que, por default, no tendrá, hasta que entremos en alguna otra función- pero somos libres de vigilar cualquier valor, argumento o variable que queramos y eso lo hacemos con *edit watches*. Para poder ver los watches, vamos a *Debugging Windows* y elegimos la correspondiente. Lo que sigue es establecer puntos de ruptura donde el compilador se detendrá para que podamos observar el proceso. Los *Break Points* se seleccionan simplemente haciendo clic a la izquierda de la línea de código y aparecen como puntos rojos, uno puede poner cuantos quiera; justo antes de que el compilador llegue a la línea con el punto rojo, se detendrá. *Toggle* o *Remove all* son funciones sobre la elección o liberación de puntos de ruptura. Podemos observar el proceso del código con un pequeño triángulo amarillo. Para seguir observando paso a paso con *Netx Line*, que le indica al compilador seguir con el código, una línea únicamente. En caso de que la siguiente línea tenga una función –o un condicional o un while o un for- entrará si los parámetros son correctos o saltará si no. Incluso si entra, no observaremos paso a paso el desarrollo en una función a menos que seleccionemos *Step Into*, que mueve el código –y los watches- a la función correspondiente; en caso de que la función sea demasiado larga y no sea necesario verificar –por ejemplo, un for con 100 iteraciones- podemos salir de la función con *Step Out*. Es así, procediendo de línea a línea, verificando en funciones y vigilando las variables, que podemos detectar cuándo y por qué hay errores en nuestro código.

Se recomiendan:

http://www.youtube.com/watch?v=XjZG6CTAsu8

http://www.youtube.com/watch?v=6BvZ91nFKEo

Tarea 1a

Implementar un algoritmo que calcule la raíz de un polinomio en un intervalo en el que los extremos tienen distinto signo.

```
#include <iostream>
#include <math.h>

using namespace std;
```

```
float f(float x){                          //función que evalúa la función dada
  return (pow(x,5)-6)*(pow(x,2)-7.5)-44;
  }

bool Zero(float a, float b){               //esta función es 1 si los extremos son de signo distinto
  if (f(a)*f(b)<0)                         //y es 0 si son del mismo signo
    return 1;
  else
    return 0;
  }

int main(){
  float a=-1, b=-0.75, m;
  bool x = Zero(a,b);
  int i=0;

  while (x && i<40){                       //si hay un 0 y van menos de 40 iteraciones
    m=(a+b)/2;                             //calcula el punto medio
    if (Zero(a, m))  b=m;                  //lo sustituye por el extremo que asegura que exista un 0
    else    a=m;
    i++;
  }

  printf("Tenemos una raíz en %f:", a);   //imprime a pantalla la raíz calculada
  return 0;
}
```

Tarea 1b

Implementar el algoritmo de Euclides para encontrar el máximo común divisor de dos números dados.

```
#include <iostream>
#include <stdio.h>

using namespace std;

int residuo(int a, int b){
  int i=0, b1=b;                           //b1 es un auxiliar que guarda el valor de b para el regreso
  while (b>=a){
    b -= a;
    i++;                                   //cuenta cuántas veces cabe a en b
  }
  return b1-(a*i);
  }
```

```
int mcd(int a, int b){
   int c;
   if (b<a){
      c=b; b=a; a=c;                    //switch. residuo requiere que entren ordenados
         }

   while(residuo(a,b)!=0){              //si el residuo es 0, uno es múltiplo del otro
   b = residuo(a, b);                   //iguala el mayor a el residuo
   if (b<a){
      c=b; b=a; a=c;         //como hace una llamada a residuo en la condición, ordena antes de salir
         }
      }
   return a;
   }

int main(){
   int a, b, MCD;
   printf("Escribe el primer entero: ");   //recibe los enteros a calcular
   scanf("%d", &a);
   printf("Escribe el segundo entero: ");
   scanf("%d", &b);

   MCD = mcd(a, b);

   printf("%d", MCD);
   return 0;
}
```

Tarea 2a

Implementar el problema 2 del examen exploratorio:

Dado float **a; y el entero int M=14;
a) Darle memoria dinámica para contener una matriz de MxM
b) Llenar toda la matriz de ceros
c) Dar el código para llenar la tridiagonal principal con números aleatorios entre 10 y 20
d) Liberar toda la memoria de la matriz

```
#include <iostream>
#include <stdio.h>
#include <time.h>

using namespace std;
```

```
int main()
{
   float **a;
   int M = 14, i, j;
   a = (float **) malloc (M*sizeof(float*));   //pide memoria dinámica para la matriz tamaño MxM
      for (i = 0; i < M; i++){
         a[i] = (float *) malloc (M*sizeof(float));
         }

   srand(time(NULL));

   for (i = 0; i < M; i++){
      for (j = 0; j < M; j++){
         if (i-j <= 1 && i-j >= -1){       //los números que cumplen esto están sobre la tri-diagonal
            a[i][j] = rand() % 10 + 10;
            }
         else a[i][j]=0;
         }
      }

   for (i = 0; i < M; i++){
      for (j = 0; j < M; j++){
         cout << a[i][j] << " ";          //imprime la matriz
            }
         cout << endl;
      }

   for (i = 0; i < M; i++){               //libera la memoria
      free(a[i]);
            }
   free(a);

   return 0;
}
```

Tarea 2b

Implementar el problema 4 del examen exploratorio:

Hacer una función que elimine los elementos repetidos de un arreglo.
Ejemplo:
Arreglo antes de llamar a la función: 3, 1, 3, 2, 2, 3, 1, 1.
Arreglo después de llamar a la función: 3, 1, 2.

```
#include <iostream>
#include <stdio.h>
#include <time.h>

using namespace std;

int * distintos(int *n, int *A){
   int aux = 1, aux2 = 0;       //aux cuenta la cantidad de enteros distintos
   int *B;           //crea un arreglo auxiliar para guardar los enteros distintos
   B = (int*) malloc ((*n)*sizeof(int));
   B[0] = A[0];       //iguala el primero
   for (int i = 1; i < *n; i++){
      for (int j = 0; j < aux; j++){  //compara cada elemento de A con los del arreglo auxiliar B
         if (A[i] != B[j])
            aux2++;}
      if (aux2 == aux){    //si alguno es igual, deja de comparar
            B[aux] = A[i];
            aux++;
            }
      aux2=0;
         }
   *n = aux;
   return B;
   }

int main()
{

   int *A, n1, n2, i;     //n2 sirve como un auxiliar para no perder el tamaño del arreglo original
   n1 = 10;
   n2 = n1;
   A = (int*) malloc (n1*sizeof(int));       //memoria pedida de manera que pueda modificarse

   srand (time(NULL));    //inicia la semilla de aleatorio

   for (int i = 0; i<n1; i++){
      A[i] = rand() % 10 + 1;     //llena el vector de datos aleatorios y lo imprime
      cout << A[i] << " ";
      }
   cout << endl;

   int *B = distintos(&n2, A);
```

```
   for (i = 0; i<n2; i++){
      cout << B[i] << " ";
      }

   free (B);
   free (A);

   return 0;
}
```

Tarea 2c

Hacer un análisis empírico del tiempo requerido para los algoritmos quickfind, quickunion y quickunion pesado.

```
#include <iostream>
#include <stdio.h>
#include <time.h>

using namespace std;

void quickfind(int *q, int *p, int *a, int n, int m){
   int t, i, j;
   for (i = 0; i < m; i++){          //m es el tamaño del arreglo de pares
      if (a[p[i]] != a[q[i]]){
         t = a[p[i]];
         for (j = 0; j < n; j++){    //n es el tamaño del vector
            if (a[j] == t)  a[j] = a[q[i]];
            }
         }
      }
   }

void quickunion(int *q, int *p, int *a, int m){
   int i, j, k;
   for (k = 0; k < m; k++){          //m es el tamaño del arreglo de pares
      for (i = p[k]; i != a[i]; i = a[i]);    //find
      for (j = q[k]; j != a[j]; j = a[j]);
      if (i == j) continue;
      a[i] = j;
      }
   }

void quickunionP(int *q, int *p, int *a, int *sz, int m){
   int i, j, k;
```

```
   for (k = 0; k < m; k++){
      for (i = p[k]; i != a[i]; i = a[i]);    //find
      for (j = q[k]; j != a[j]; j = a[j]);
      if (i == j) continue;
      if (sz[i] < sz[j]){
         a[i] = j; sz[j] += sz[i];
         }
      else{
         a[j] = i; sz[i] += sz[j];
         }
      }
   }

int main()
{

   int *p, *q, *A, *B, *C, *sz, i, arr, par;
   arr  = 1000000;              //tamaño del arreglo
   par = 100000;                //cantidad de pares
   float t1;
   srand(time(NULL));
   clock_t start, finish;

   A = (int *) malloc (arr*sizeof(int));      //hace un arreglo para cada algoritmo
   B = (int *) malloc (arr*sizeof(int));
   C = (int *) malloc (arr*sizeof(int));
   sz = (int *) malloc (arr*sizeof(int));
   q = (int *) malloc (par*sizeof(int));
   p = (int *) malloc (par*sizeof(int));

   for (i = 0; i < arr; i++){              //los tres algoritmos tienen los mismos datos
      A[i] = i;
      B[i] = i;
      C[i] = i;
      sz[i] = 1;
      }

   for (i = 0; i < par; i++){              //genera los pares p-q en dos arreglos
      q[i] = rand() % par;
      p[i] = rand() % par;
      }

   start = clock();
   quickfind(q, p, A, arr, par);
```

```
   finish = clock();
   t1 = ((finish-start)/CLOCKS_PER_SEC);
   cout << "Quick find. Tamaño del arreglo: " << arr << " Cantidad de parejas: " << par << " Tiempo:
" << t1 << endl;

   start = clock();
   quickunion(q, p, B, par);
   finish = clock();
   t1 = ((finish-start)/CLOCKS_PER_SEC);
   cout << "Quick union. Tamaño del arreglo: " << arr << " Cantidad de parejas: " << par << "
Tiempo: " << t1 << endl;

   start = clock();
   quickunionP(q, p, C, sz, par);
   finish = clock();
   t1 = ((finish-start)/CLOCKS_PER_SEC);
   cout << "Quick union. Tamaño del arreglo: " << arr << " Cantidad de parejas: " << par << "
Tiempo: " << t1 << endl;

   return 0;
}
```

Tabla de tiempo

QF = Quick Find QU = Quick Union QUP = QU Pesado

Tamaño del arreglo: 1,000,000

Cantidad de Pares	Tiempo QF	Tiempo QU	Tiempo QUP
100	0	0	0
1000	4	0	0
10000	39	0	0
100000	151	5	0

Tamaño del arreglo: 10,000,000

1000	38	0	0
10000	378	0	0
100000	1510	5	0
1000000	NA	90	0

Tamaño del arreglo: 100,000,000

10000	NA	0	0
100000	NA	5	0
1000000	NA	91	0
10000000	NA	948	0

QF llegó a tardar 25 minutos algo que le tomó a QUP 0 segundos. Lo mismo para 15 minutos de QU contra 0 segundos de QUP. definitivamente QUP es mucho más eficiente que QF.

Práctica 1

Hacer una función que evalúe un polinomio donde todos los coeficientes los recibe de consola, igual que el valor a evaluar. OBSERVACIÓN: este código genera un loop infinito.

```
#include <iostream>
#include <math.h>
#include <cstdio>
#include <stdlib.h>

using namespace std;

double poly(double x, double *coefs, int grado){
   double valor=0;
   for (int i=0; i<grado+1; i++){
      valor += coefs[i]*pow(x, i);
      }
   return valor;
}

int main()
{

   int grado=0;
   double *coefs;
   double x=0;

   printf("Buenos días \n");
   printf("Introduce el grado del polinomio: \n");
   scanf("%d", &grado);                    //guarda en grado el grado del polinomio
   coefs = (double*) calloc(grado+1, sizeof(double));      //pide memoria para el arreglo

   printf("Introduce los coeficientes: \n");
   for (int i=0; i<grado+1; i++){
      printf("a%d:", i);
      scanf("%lf", &coefs[i]);       //guarda los valores de los coeficientes
   }

   while(1){                          //loop infinito. no sé cómo romper con teclas
      printf("Introduce el valor x a evaluar \n");
      scanf("%lf", &x);
      printf("El valor del polinomio en %lf es: %lf \n", x, poly(x, coefs, grado));
   }
   return 0;
}
```

Práctica 2

1. Escribir un programa que le pida al usuario un número x. Lo lea. Internamente llame a una función de tipo void en la cual reemplaza x por sen(x). Fuera de función, imprime el número a pantalla.
2. Escribir un programa que el pida al usuario un número x. Lo lea e imprima su dirección. Internamente llame a una función de tipo void en la cual el apuntador leído sea reemplazado por NULL. Fuera de función, imprime el apuntador a pantalla.
3. Escribir un programa que haga la alocación de memoria dinámica para una matriz de reales de mxn, con doble apuntador. Las dimensiones se piden al usuario. Rellenar la matriz con números pseudo aleatorios en el intervalo [a,b] que también se pide al usuario. Hacer otra función que reciba la matriz A, la convierta en A transpuesta y regrese un bool de si es o no cuadrada. Mostrar en pantalla A, A transpuesta y si es o no cuadrada. Finalmente, liberar memoria.

OBSERSVACIÓN: este código, tal cual está, genera un segmentation fault.

```
#include <iostream>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define PI 3.14159265   //será necesario porque sin recibe el parámetro en radianes

using namespace std;

void seno(double &x){
   x = sin(x*PI/180);  //convierte a radianes en el argumento
   }

void nulificaAp(double &x){
   x = NULL;
   }

void matriz(int columnas, int renglones, float ***M){
   int i;
   M = new float**[renglones];
   for (i = 0; i < columnas; i++){
      M[i] = new float*[columnas];
      }
   }

void llenaMatriz(int columnas, int renglones, float a, float b, float ***M){
   srand(time(NULL));
```

```
   for (int i = 0; i < columnas; i++){
      for (int j = 0; j < renglones; j++){
         *M[i][j] = (rand() % 10^6 + 1)*(b-a)/(10^6) + a;
         }
      }
   }

bool cuadrada(int columnas, int renglones){
   if (columnas == renglones)
      return 1;
   else       return 0;
   }

void imprimeTranspuesta(int columnas, int renglones, float ***M){
   for (int i = 0; i < columnas; i++){
      for (int j = 0; j < renglones; j++){
         cout << " " << M[j][i];
         }
      cout << endl;
      }
   }

void destruyeMatriz(int columnas, int renglones, float ***M){
   int i;
   for (i = 0; i < columnas; i++){
      delete M[i];
      }
   delete M;
   }

int main()
{

  double x;

   cout << "Escribe el valor de x: " << endl;
   cin >> x;
   seno(x);
   cout << "El seno de x es: " << x << endl << endl;

   cout << "Escribe otro número: " << endl;
   cin >> x;
   cout << &x;
```

```
    nulificaAp(x);
    cout << "Apuntador nulo: " << x << endl <<endl;

    int col, ren, i, j;
    float **A, a, b;
    cout << "Escribe el número de columnas: " << endl;
    cin >> col;
    cout << "Escribe el número de renglones: " << endl;
    cin >> ren;
    cout << "Escribe el mínimo de los valores en la matriz: " << endl;
    cin >> a;
    cout << "Escribe el máximo de los valores en la matriz: " << endl;
    cin >> b;

    matriz(col, ren, &A);
    llenaMatriz(col, ren, a, b, &A);
    for (i = 0; i < ren; i++){
        for (j = 0; j < col; j++){
            cout << " " << A[i][j];
        }
        cout << endl;
    }

    if (cuadrada(col, ren)){
        cout << "La matriz es cuadrada. Su transpuesta es: " << endl;
        imprimeTranspuesta(col, ren, &A);
        }
    else{
        cout << "La matriz no es cuadrada. Su transpuesta: " << endl;
        imprimeTranspuesta(col, ren,&A);
        }

    return 0;
}
```