Estructura de datos y algoritmos

Capítulo 1 - Introducción

- Principio DRY (Don't repeat yourself)
  - Si hay alguien que ya lo ha hecho, no repetir.
  - Ya está probado, y hay una comunidad de usuarios que lo utilizan.
  - ¿Lo podemos hacer mejor?
  - Race conditions.

Algoritmo

> Conjunto finito y ordenado de instrucciones para resolver un problema dado

En definitiva: una receta

Ejemplos

- Calcular el mayor elemento de un array
- Ordenar una serie de números
- Calcular el camino óptimo para llegar del punto A al B…

Qué se le debe pedir:

- Correcto
- Eficaz
- Fácil de entender
- Haga uso eficiente de los recursos

Método de trabajo

- Se diseña un algoritmo sencillo
- Se hacen simulaciones y mediciones
- Se plantea un algoritmo más eficiente
- Análisis de mejora

Porqué estudiar algoritmos:

- Importante para todas las ramas de la computación
    i. Redes (optimización de caminos)
    ii. Diseño (algoritmos de geometría)
    iii. Bases de datos (b-trees)
    iv. Criptografía
    v. ...
- Importante en la innovación tecnológica
- Nos hace más inteligentes (otra forma de resolver problemas)

Porqué estudiar algoritmos:

- Mejora nuestras habilidades Matemáticas
- Te ayuda a posibles entrevistas
- Comienzas a pensar de "Algortímicamente"
- Nos hace mejores programadores

Demostración

- Page Rank
- Apple compra SnappyLabs, la empresa creadora de SnappyCam (5-1-2014)

Corrección (correspondería a otra asignatura)

- Componente matemático muy fuerte
- Se hace a través de:
    i. Precondiciones
    ii. Postcondiciones
    iii. Invariantes

Ejemplo

Calcular el mayor elemento de un array

```
int max(int n,const int a[n]){
        int m = a[0];
        int i =1;
        while(i != n){
                if(m < a[i])
                m = a[i];
                ++i;
        }
        return m;
}
```

Demostración

el programa es correcto:

```
int max(int n,const int a[n]){
        int m = a[0];
        // m es el mayor número en el subarray a[0...0]
        int i =1;
        while(i != n){
                // m es el mayor número en el subarray a[0...i-1]
                if(m < a[i])
                        m = a[i];
                        // m es el mayor número en el subarray a[0...i]
                ++i;
                // m es el mayor número en el subarray a[0...i-1]
        }
        // m es el mayor número en el subarray a[0...i-1], & i==n
        return m;
}
```

Análisis de un programa

Frente al rendimiento

- Memoria
- Tiempo de ejecución
- Otros recursos

Otros análisis

- Los recursos pueden estar fijados
  1. ¿Tamaño máximo que puedo resolver en T secs.?
  2. ¿Tamaño máximo con M megs?
- Referencias:

  Cyberpunk 2077

Tiempo de ejecución de un programa

¿Cómo se puede calcular?

> Contando instrucciones y ponderando por el tiempo de ejecución de cada instrucción

Ejemplo 1

```
// Inicialización
int max(int n,constint a[n]){
        int m = a[0];
        int i =1;
        while(i != n){
                if(m < a[i])
                m = a[i];
                ++i;
        }
        return m;
}
```

Factores de los que depende

- Calidad del código que genera el compilador
- Instrucciones de la máquina
- Velocidad de la máquina
- Datos de entrada de un programa
    i. No dependen de la entrada (A veces)
    ii. Sólo dependen del tamaño del problema (Habitualmente)

Casos mejor, peor, promedio

Por ejemplo, si el código tiene un condicionante en su ejecución, no se puede calcular el tiempo de ejecución sin saber los datos de entrada.

> Por ejemplo, determinar si existe un elemento en un array

```
for (int i = 0; i < sizeof(output); i++)
{
      if (input [count] == output[i][])
      {

            //.......
                        break;
      }

}
```

¿Tiene mucho sentido?

No es un tiempo exacto, porque:

- Depende del compilador
- Depende del tipo de máquina
- Depende la velocidad de la máquina
- Compite con los recursos del ordenador

¿Y los Algoritmos?

No se ejecutan en ninguna máquina

> Rendimiento de un algoritmo = Complejidad

Notaciones

Se dice que:

T(n) = O(n²)

Si existen constanstes (n₀, c) positivas que:

T(n)≤ c * n ²

para n≥n₀

Notación Ω

Big O = cota superior

Ω = cota inferior

Notación θ

$T(n)=\theta(f(n))$

Sí y sólo si:

$T(n)=O(f(n))$ && $T(n)=\Omega(f(n))$

Origen

Donald Knuth. "Big Omicron and big Omega and big Theta", SIGACT News, Apr.-June 1976, 18-24

Teoremas

Teorema: Si $T(n) = a_k n^k + \ldots + a_1 n + a_0$ Entonces

$$T(n) = O(n^k)$$

Teorema: Para cada $k \geq 1$, n^k no es $O(n^{k-1})$

Términos Dominantes

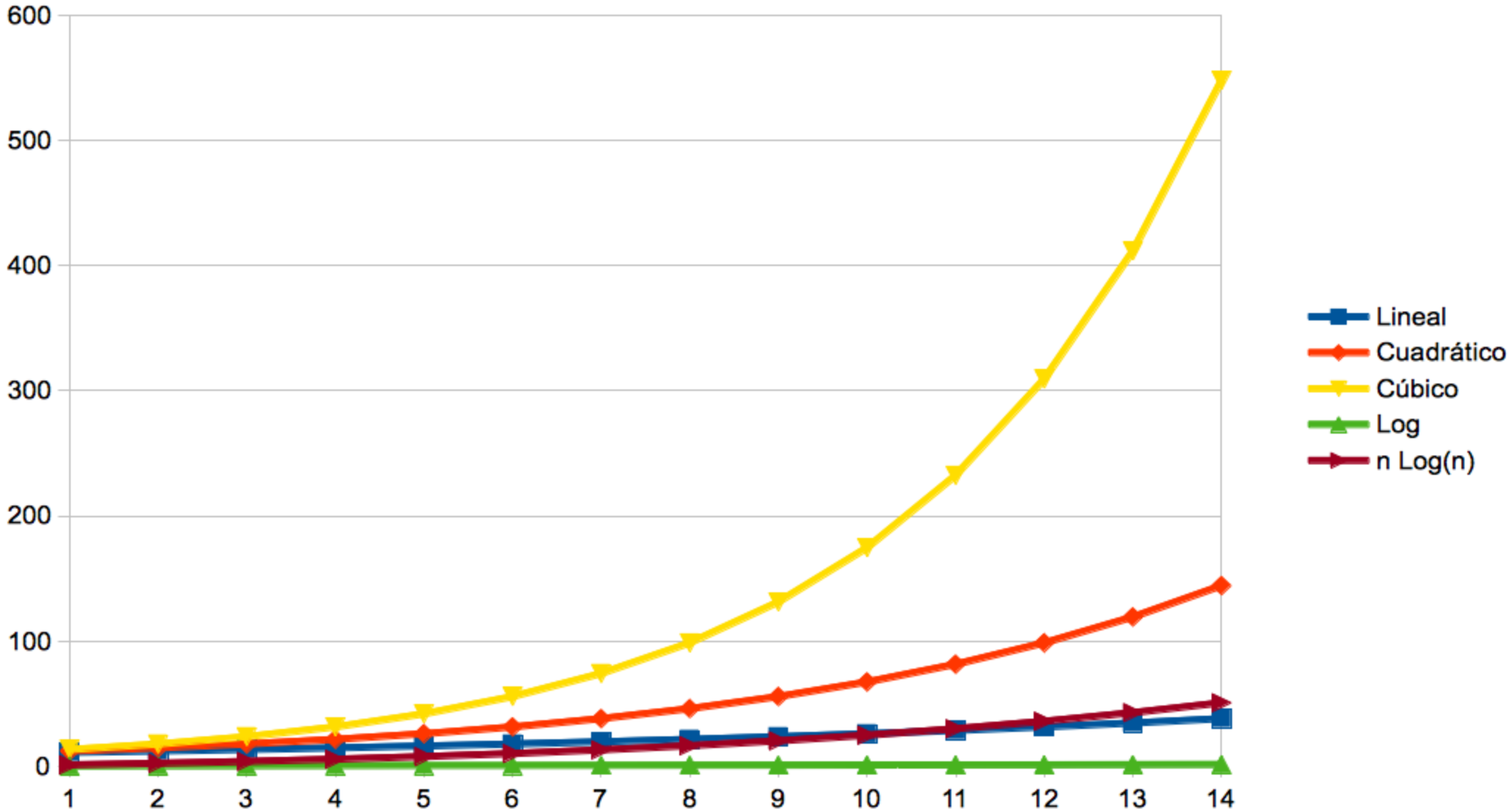
Suponer que estimamos 350 N²+ N + N ³ por N ³.

Para N = 10000:

- El valor actual es 1,003,500,010,000
- La estimación es 1,000,000,000,000

El error en la estimación es de 0.35%, lo que influye bien poco.

Rendimiento



Conclusiones

> Para N grande, el término dominante es usualmente indicativo del comportamiento del algoritmo.

> Para N pequeño, el término dominante no es necesariamente indicativo del comportamiento del algoritmo, PERO, Los programas típicos con datos de entrada pequeños se ejecutan tan rápìdo que no nos preocupamos de ello.

Ejemplo 2

PUNTOS CERCANOS EN EL PLANO

Dados N puntos en el plano (Sistema de coordenadas x-y) encontrar el par de puntos que están mas cercanos entre sí.

1. Calcular la distancia entre cada par de puntos
2. Retener la mínima distancia

N * (N-1) / 2

EL algoritmo es cuadrático

Ejemplo 3

PUNTOS COLINEALES EN EL PLANO

Dados N puntos en el plano, determinar si existen tres que formen una línea

1. Enumeramos todos los grupos de tres puntos.
2. Por cada grupo chequeamos si forman línea.

N(N-1)(N-2) / 6. Es un algoritmo cúbico; insostenible para 10,000 puntos.

Se conoce un algoritmo cuadrático. ¿Es mejor?.

paper

Ejemplo 4

Busqueda Binaria

Mirar en la mitad del array

- Si X es menor que el elemento del medio, entonces buscar en el subarray a la izquierda de la mitad
- Si X es mayor que el elemento del medio, entonces buscar en el subarray a la derecha de la mitad
- Si X es igual al elemento de la mitad, entonces ¡ encontrado !
- Si el array es vacio, el elemento no está.

Código

```
template <class Comparable>
int binarySearch( const vector<Comparable> & a, const Comparable & x )
{
    int inf = 0;
    int sup = a.size( ) - 1;
    int med;

    while( inf <= sup )
    {
        med = ( inf + sup ) / 2;

        if( a[ med ] < x )
            inf = med + 1;
        else if (a[med] > x)
            sup = med -1;
        else
            return med;
    }
    return NOT_FOUND;
}
```

Ejemplo 3

Merge Sort

24,56,12,34,3,78,32,9

Dividimos el array ...

24,56,12,34 3,78,32,9

... Recursivamente ...

24,56 12,34 3,78 32,9

Ordenamos cada parte

24,56 12,34 3,78 9,32

Mezclamos el array

24,56 + 12,34 3,78 + 9,32

12,24,34,56 + 3,9,32,78

Implementación 1

```
void mergesort(int *a, int*b, int low, int high)
{
    int pivot;
    if(low<high) {
        pivot=(low+high)/2;
        mergesort(a,b,low,pivot);
        mergesort(a,b,pivot+1,high);
        merge(a,b,low,pivot,high);
    }
}
```

Implementación 2

```
void merge(int *a, int *b, int low, int pivot, int high)
{
    int h,i,j,k;
    h=low;
    i=low;
    j=pivot+1;
    while((h<=pivot)&&(j<=high)) {
        if(a[h]<=a[j]) {
            b[i]=a[h];
            h++;
        }
        else {
            b[i]=a[j];
            j++;
        }
        i++;
    }
    if(h>pivot) {
        for(k=j; k<=high; k++) {
            b[i]=a[k];
            i++;
        }
    }
    else {
        for(k=h; k<=pivot; k++) {
            b[i]=a[k];
            i++;
        }
    }
    for(k=low; k<=high; k++) a[k]=b[k];
}
```

Implementación 3

```
int main()
{
    int a[] = {12,10,43,23,-78,45,123,56,98,41,90,24};
    int num;

    num = sizeof(a)/sizeof(int);

    int b[num];

    mergesort(a,b,0,num-1);

    for(int i=0; i<num; i++)
        cout<<a[i]<<" ";
    cout<<endl;
}
```

Análisis del Algoritmo

T(N) = el tiempo del algoritmo en resolver un problemas de tamaño N.

Entonces T(1) = 1 (1 será una unidad de tiempo; recordar que no importan las constantes)

T(N) = 2 T(N / 2) + N

- Dos llamadas recursivas, cada una de tamaño N/2. El tiempo de resolver cada llamada recursiva es T(N / 2).
- El tercer Caso toma tiempo O(N); usamos N, porque quitamos las constantes.

Expandiendo la recurrencia

$T(1) = 1 = 1 * 1$

$T(2) = 2 * T(1) + 2 = 4 = 2 * 2$

$T(4) = 2 * T(2) + 4 = 12 = 4 * 3$

$T(8) = 2 * T(3) + 8 = 32 = 8 * 4$

$T(16) = 2 * T(4) + 16 = 80 = 16 * 5$

$T(32) = 2 * T(5) + 32 = 192 = 32 * 6$

$T(64) = 2 * T(6) + 64 = 448 = 64 * 7$

$T(N) = N(1 + \log N) = O(N \log N)$

Ejemplo 6

Máxima Suma en Secuencias Contiguas

Dados (puede haber enteros negativos) A₁, A₂, …, A_N,

encontrar (e identificar la secuencia correspondiente) el máximo valor de (A_i + A_{i+1} +…+ A_j).

-1, 3, -5, 4, 6, -1, 2, -7, 13, -3

Método 1

```
template <class Comparable>
Comparable maxSubsecuenciaSum1( const vector<Comparable> & a, int & iniSec,
                                        int & finSec ){
        int n = a.size( );
        Comparable maxSum = 0;
        Comparable SumParcial = 0;
        for( int i = 0; i < n; i++ )
                for( int j = i; j < n; j++ ){
                        SumParcial = 0;
                        for( int k = i; k <= j; k++ )
                                        SumParcial += a[ k ];
                        if( SumParcial > maxSum ){
                                maxSum = SumParcial;
                                iniSec = i;
                                finSec = j;
                        }
                }
        return maxSum;
}
```

¿Mejorable?

Sí:

Borrando un bucle (no es siempre posible).

El bucle interno es innecesario ya que repite cálculos innecesarios.

SumParcial para el siguiente j es fácilmente obtenible desde el anterior valor de SumParcial:

Necesitamos A_i + A_{i+1} +...+ A_{j-1}+ A_j

Cálculos ya realizados Ai +A i+1 + ...+ A j-1

Lo que necesitamos es lo anterior + Aj

Método 2

```
template <class Comparable>
Comparable maxSubsecuenciaSum2( const vector<Comparable> & a,
                                int & seqStart, int & seqEnd )
{
        int n = a.size( );
        Comparable maxSum = 0;
        for( int i = 0; i < n; i++ ){
                Comparable thisSum = 0;
                for( int j = i; j < n; j++ ){
                        thisSum += a[ j ];
                        if( thisSum > maxSum ){
                                maxSum = thisSum;
                                seqStart = i;
                                seqEnd = j;
                        }
                }
        }
        return maxSum;
}
```

¿Mejorable?

Sí:

Usamos la técnica de Divide y Venceras

Máxima Suma en Secuencias Contiguas

Buscar en la primera mitad.
Buscar en la segunda mitad
Comenzar en algún elemento de la primera mitad, ir hasta el último elemento de la primera mitad, continuar en el primer elemento de la segunda mitad y terminar en algún elemento de la segunda mitad.

Calcular las tres posibilidades y quedarse con el máximo.
Las primeras dos posibilidades son fácilmente computables usando recursión.

Método 3

```
template <class Comparable>
Comparable maxSubSum( const vector<Comparable> & a, int left, int right )
{
    Comparable maxLeftBorderSum = 0, maxRightBorderSum = 0;
    Comparable leftBorderSum = 0, rightBorderSum = 0;
    int center = ( left + right ) / 2;
    if( left == right )         // Caso Base.
        return a[ left ] > 0 ? a[ left ] : 0;
    Comparable maxLeftSum  = maxSubSum( a, left, center );
    Comparable maxRightSum = maxSubSum( a, center + 1, right );
    for( int i = center; i >= left; i-- ){
        leftBorderSum += a[ i ];
        if( leftBorderSum > maxLeftBorderSum )
            maxLeftBorderSum = leftBorderSum;
    }
    for( int j = center + 1; j <= right; j++ ){
        rightBorderSum += a[ j ];
        if( rightBorderSum > maxRightBorderSum )
            maxRightBorderSum = rightBorderSum;
    }
        return max3( maxLeftSum, maxRightSum,maxLeftBorderSum + maxRightBorderSum );
}
```

Método 4

¿Se puede generar un algoritmo mejor?

Sí, ¡ De orden lineal !

Enlace externo

Código

```
template <class Comparable>
Comparable maxSubsecuenciaSum4( const vector<Comparable> & a,
                                int & iniSec, int & finSec )
{
    int n = a.size( );
    Comparable sumaParcial = 0;
    Comparable maxSum = 0;
    for( int i = 0, j = 0; j < n; j++ )    {
        sumaParcial += a[ j ];
        if( sumaParcial > maxSum ){
            maxSum = sumaParcial;
            iniSec = i;
            finSec = j;
        }
        else if( sumaParcial < 0 ){
            i = j + 1;
            sumaParcial = 0;
        }
    }
    return maxSum;
}
```

Tiempos de ejecución

- Algoritmo #4 N = 1 tiempo = 0.000000
- Algoritmo #3 N = 1 tiempo = 0.000000
- Algoritmo #2 N = 1 tiempo = 0.000000
- Algoritmo #1 N = 1 tiempo = 0.000000
- Algoritmo #4 N = 10 tiempo = 0.000000
- Algoritmo #3 N = 10 tiempo = 0.000001
- Algoritmo #2 N = 10 tiempo = 0.000001
- Algoritmo #1 N = 10 tiempo = 0.000002
- Algoritmo #4 N = 100 tiempo = 0.000001
- Algoritmo #3 N = 100 tiempo = 0.000010
- Algoritmo #2 N = 100 tiempo = 0.000028
- Algoritmo #1 N = 100 tiempo = 0.000672

Tiempos de ejecución

- Algoritmo #4 N = 1000 tiempo = 0.000007
- Algoritmo #3 N = 1000 tiempo = 0.000104
- Algoritmo #2 N = 1000 tiempo = 0.002339
- Algoritmo #1 N = 1000 tiempo = 0.500000
- Algoritmo #4 N = 10000 tiempo = 0.000059
- Algoritmo #3 N = 10000 tiempo = 0.001214
- Algoritmo #2 N = 10000 tiempo = 0.222222
- Algoritmo #1 N = 10000 tiempo = 590.000000
- Algoritmo #4 N = 100000 tiempo = 0.000585
- Algoritmo #3 N = 100000 tiempo = 0.013605
- Algoritmo #2 N = 100000 tiempo = 23.000000

Conclusión

> "Perhaps the most important principle for the good algorithm designer is to refuse to content"

Aho, Hopcroft, Ullman The design and Analysis of computer Algorithms (1974)