Temario

- Repaso de C++.
  - Entender ejercicios
  - Declaración de variables
  - Ámbito de una variable
  - Pila
  - Funciones. Call stack
  - Buenas prácticas
  - Punteros tradicionales

Algoritmo

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

Tipos (I)

- Algoritmos deterministas: Para los mismos datos de entrada se producen los mismos datos de salida.
- Algoritmos no deterministas: Para los mismos datos de entrada pueden producirse diferentes de salida.

Tipos (II)

- Recursivos
  - De forma directa o indirecta se llama a sí mismo
- Iterativos
  - Utiliza bucles

Análisis de Algoritmos

Búsqueda de un elemento en un array

- Mejor caso. Se encuentra x en la 1ª posición:
  - `Tiempo(N) = a`
- Peor caso. No se encuentra x:
  - `Tiempo(N) = b·N + c`
- Caso medio. Se encuentra x con probabilidad P:
  - `Tiempo(N) = b·N + c - (d·N+e)·P`

Notaciones

Se dice que:

$f(n) \in O(g(n))$
si existen números positivos c y N tales que $f(n) \leq c\, g(n)\ \forall\ n \geq N$.

Notación Ω

Se dice que:

$f(n) \in \Omega(g(n))$

si existen números positivos c y N tales que $f(n) \geq c\, g(n)\ \forall\ n \geq N$.

Notación Θ

$f(n) \in \Theta(g(n))$

si existen números positivos c_1, c_2, y N tales que

$c_1\, g(n) \leq f(n) \leq c_2\, g(n) \;\forall\; n \geq N.$

$f(n) \in \theta(g(n))$

Sí y sólo si:

$f(n) \in O(g(n)) \;\&\&\; f(n) \in \Omega(g(n))$

Origen

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

Algunas reglas básicas:

- Operaciones básicas (+, -, *, :=,...): Una unidad de tiempo, o alguna constante.
- Operaciones de entrada salida: Otra unidad de tiempo, o una constante diferente.
- Bucles FOR: Se pueden expresar como un sumatorio, con los límites del FOR.
- IF y SWITCH: Estudiar lo que puede ocurrir. Mejor caso y peor caso según la condición. ¿Se puede predecir cuándo se cumplirán las condiciones?
- Llamadas a procedimientos: Calcular primero los procedimientos que no llaman a otros.
- Bucles WHILE y DO WHILE: Estudiar lo que puede ocurrir. ¿Existe una cota inferior y superior del número de ejecuciones? ¿Se puede convertir en un FOR?

```
int fibonacciIterativo(int input) {
        if (input <= 2) return (1);
        int ult=1, penult=1;
        int result = 0;
        for (auto j = 2; j < input; ++j) {
                result = ult + penult;
                penult = ult;
                ult = result;
        }
        return result;
}
```

```
Hanoi (N, A, B, C: integer)
        if  N=1 then
              Mover (A, C)
        else begin
              Hanoi (N-1, A, C, B)
              Mover (A, C)
              Hanoi (N-1, B, A, C)
        end
```

Ejemplos bucles

```
        // Inicialización
        for (int i=0;i<n;++i>){
    // algo constante
        }
```

Ejemplos bucles (II)

```
        // Inicialización
        for (int i=0;i<n;++i>){
            for (int j=0;j<n;++j>){
            // algo constante
        }
        }
```

Teoremas

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

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

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

Ejercicio propuesto 1

Fibonacci

- Crear una versión iterativa y otra recursiva.
- Comparar tiempos

```
int fibonnaci(int i) {
        //cout <<i<<endl;
        if (i < 2) return (1);
        else return (fibonnaci(i - 1) + fibonnaci(i - 2));
}
```

```
int fibonnaciIterative(int i) {
        if (i <= 2) return (1);
        int ult=1, penult=1;
        int result = 0;
        for (auto j = 2; j < i; ++j) {
                result = ult + penult;
                penult = ult;
                ult = result;

        }
        return result;

}
```

Benchmark	Time	CPU	Iterations
testFibonnaci/0	3.33 ns	3.35 ns	224000000
testFibonnaci/5	29.2 ns	29.2 ns	23578947
testFibonnaci/10	310 ns	307 ns	2240000
testFibonnaci/15	3429 ns	3453 ns	203636
testFibonnaci/20	38633 ns	38365 ns	17920
testFibonnaci/25	435077 ns	439551 ns	1493
testFibonnaci/30	4612186 ns	4614094 ns	149

Benchmark	Time	CPU	Iterations
testFibonnaciIter/0	3.14 ns	3.14 ns	224000000
testFibonnaciIter/5	4.19 ns	4.20 ns	160000000
testFibonnaciIter/10	6.59 ns	6.56 ns	112000000
testFibonnaciIter/15	9.00 ns	9.00 ns	74666667
testFibonnaciIter/20	8.94 ns	8.89 ns	89600000
testFibonnaciIter/25	12.8 ns	12.8 ns	56000000
testFibonnaciIter/30	13.6 ns	13.7 ns	56000000

Términos Dominantes

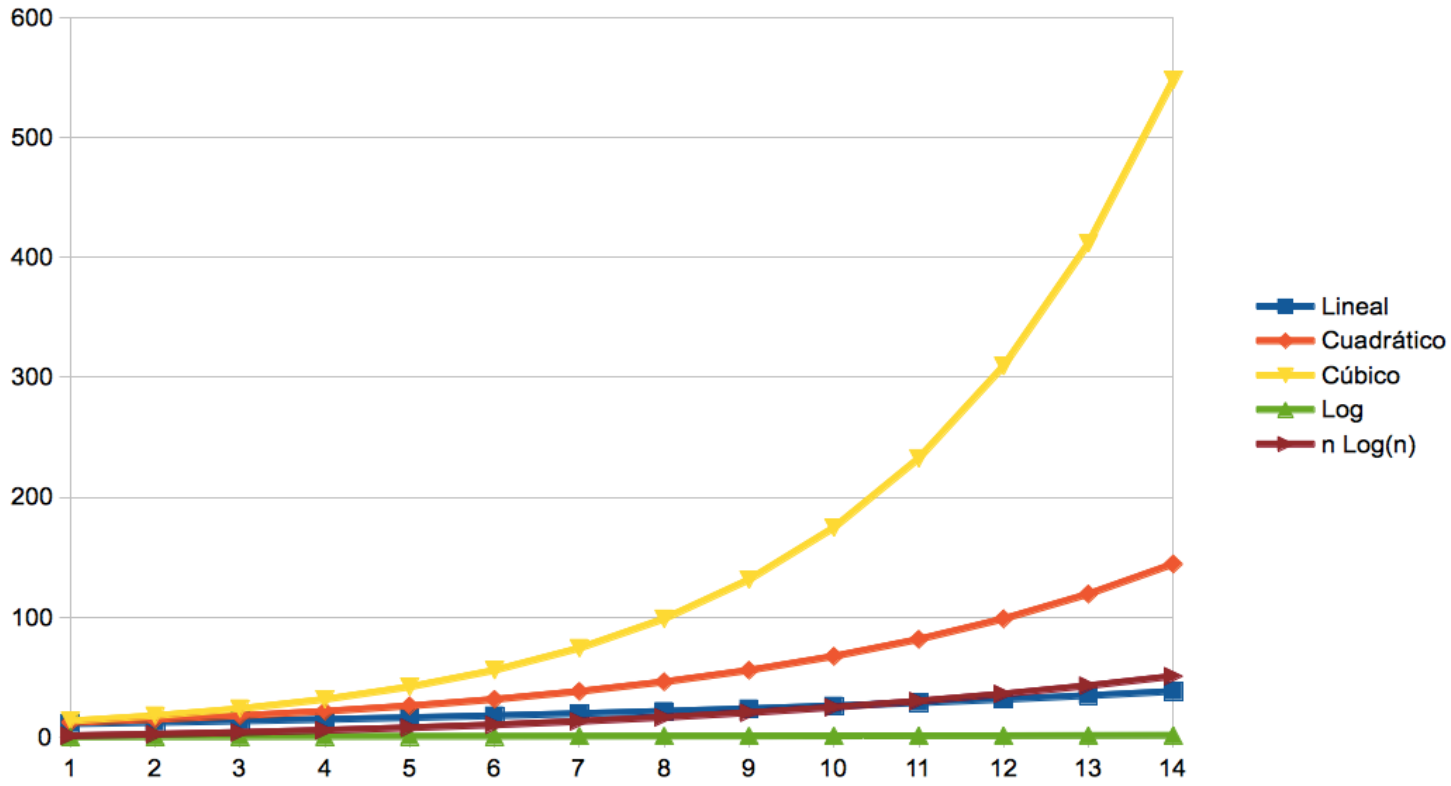
Suponer que estimamos $350\ N^2 + N + N^3$ por N^3.

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.

- Cubica: el término dominante es unas cuantas veces (constante) N3. Decimos O(N 3).
- Cuadrática: el término dominante es unas cuantas veces (constante) N2. Decimos O(N 2).
- O(N log N): el término dominante es unas cuantas veces N log N.
- Lineal: el término dominante es unas cuantas veces N . Decimos O(N).
- Constante: no depende de la entrada O(1)
  Ejemplo: $350N^2 + N + N^3$ es cúbica

MÍNIMO ELEMENTO EN UN ARRAY

Dado un array de N elementos, encontrar el mínimo

```
1.- Mantener una variable min que guarde el mínimo elemento.
2.- Inicializar min al primer elemento
3.- Realizar una Búsqueda Secuencial en el array y actualizar min cuando corresponda
```

Obviamente el algoritmo es una búsqueda Secuencial.
El tiempo de ejecución de este algoritmo será lineal O(N) ya que repetimos una cantidad fija de trabajo por cada elemento del array.
Este Algoritmo Lineal es tan bueno como nosotros esperábamos ya que tenemos que examinar cada elemento del array, que es un proceso que requiere tiempo lineal.

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

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;
}
```

O(log(n))

Ejemplo

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.

- Cualquier algoritmo recursivo que se resuelva dividiendo el problema en dos subproblemas de tamaño de entradas iguales y tenga un trabajo lineal No Recursivo para combinar las dos soluciones siempre tendrá tiempo de ejecución O(N log N) ya que el análisis anterior siempre será válido.
- Esta es una mejora significativa en cuanto al algoritmo cuadrático, pero no supera al O(N), aunque hemos visto que no es muy intuitivo.

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) A_1, A_2, …, 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 $A_i + A_{i+1} + \ldots + A_{j-1} + A_j$

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 !

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)

Práctica

- Función que devuelva el mayor elemento de un array.
- Función que devuelva el mayor elemento de un array (ordenado).
- Ordenar un array

Suma contigua:

- 1er método
- 2o método
- ¿lineal?