# INF221 – Algoritmos y Complejidad

# Clase #17
# Dividir y Conquistar II

Aldo Berrios Valenzuela Horst H. von Brand

Miércoles 5 de octubre de 2016

## 1. Dividir y Conquistar II

Discutiremos un problema planteado por Bentley [1]. Dado el arreglo $a[n]$, hallar la máxima suma de un rango:

$$\max_{i,j}\left\{\sum_{i\le k\le j} a[k]\right\} \tag{1.1}$$

Si todos los valores son positivos, la respuesta es obvia: la suma de todos los elementos del arreglo. El punto está si hay elementos negativos: ¿incluimos uno de ellos en la esperanza que los elementos positivos que lo rodean más que lo compensen? Finalmente, acordamos que la suma de un rango vacío es cero, y que en un arreglo de elementos negativos la suma máxima es cero.

### 1.1. Algoritmo ingenuo

La solución obvia, traducción directa de la especificación ecuación (1.1), es la mostrada en el listado 1. La comple-

```
double MaxSum(double a[], int n)
{
  double MaxSoFar;

  MaxSoFar = 0.0;
  for(int i = 0; i < n; i++) {
    for(int j = i; j <= n; j++) {
      double Sum = 0.0;
      for(int k = i; k < j; k++)
        Sum += a[k];
      MaxSoFar = max(MaxSoFar, Sum);
    }
  }
  return MaxSoFar;
}
```

Listado 1: Algoritmo 1: Versión ingenua

jidad del algoritmo 1 es $O(n^3)$. Lo que buscamos es mejorarlo.

### 1.2. No recalcular sumas

Hay dos ideas sencillas para evitar recalcular sumas.

### 1.2.1. Extender sumas

En vez de calcular la suma del rango cada vez, extendemos la suma anterior. Esto da el programa del listado 2. La

```
double MaxSum(double a[], int n)
{
  double MaxSoFar;

  MaxSoFar = 0.0;
  for(int i = 0; i < n; i++) {
    double Sum = 0.0;
    for(int j = i; j < n; j++) {
      Sum += a[j];
      MaxSoFar = max(MaxSoFar, Sum);
    }
  }
  return MaxSoFar;
}
```

Listado 2: Algoritmo 2: Evitar recalcular sumas

complejidad del algoritmo 2 es $O(n^2)$.

### 1.2.2. Sumas cumulativas

Una manera de manejar rangos es usar sumas cumulativas, y obtener el valor para el rango restando. Esta idea da el listado 3. La complejidad del algoritmo 3 es $O(n^2)$. En función de nuestro algoritmo original resulta una mejora,

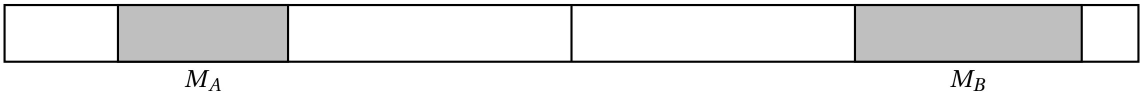
```
double MaxSum(double a[], int n)
{
  double CumArray[n + 1]; /* Sum of a[0] to a[i - 1] */
  double MaxSoFar;

  CumArray[0] = 0.0;
  for(int k = 0; k < n; k++)
    CumArray[k + 1] = CumArray[k] + a[k];

  MaxSoFar = 0.0;
  for(int i = 0; i < n; i++)
    for(int j = i; j < n; j++)
      MaxSoFar = max(MaxSoFar,
                     CumArray[j + 1] - CumArray[i]);

  return MaxSoFar;
}
```

Listado 3: Algoritmo 3: Usar arreglo cumulativo

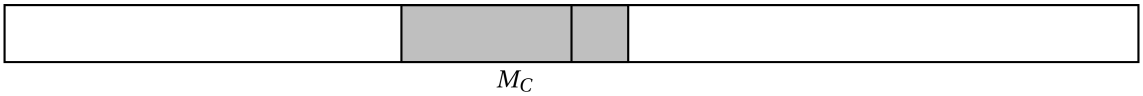
pero no respecto a la segunda variante.

## 1.3. Dividir y Conquistar

Aplicar la estragia vista la clase pasada lleva a la figura 1a. Pero debemos también considerar que el rango con máxima suma esté a hojarcadas, cruzando el punto central, como en la figura 1b. El algoritmo es el dado en el listado 4. Usando el "teorema maestro", las constantes del algoritmo 4 son $a = 2$, $b = 2$, $d = 1$, por lo tanto la complejidad es $O(n \log n)$.




(a) Máximos en las mitades




(b) Máximo al medio

Figura 1: Dividir y conquistar

### 1.4. Un algoritmo lineal

Otro algoritmo resulta de la idea, común al procesar arreglos, de tener una solución parcial hasta $a[i]$, y analizar cómo extenderla para cubrir hasta $a[i+1]$. En nuestro caso, esto significa considerar la máxima suma que llega hasta $a[i]$, y recordar la máxima suma vista hasta ahora, ver la figura 2. Esto da el algoritmo 5, del listado 5.

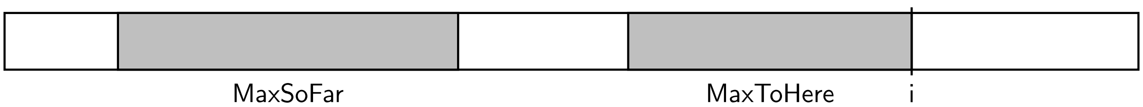



Figura 2: Extender la solución

La complejidad del algoritmo 5 es $O(n)$. Sin embargo, es imposible tener una complejidad menor que $n$, dado que es necesario revisar cada elemento del arreglo.

| **Algoritmo** | **1** | **2** | **4** | **5** |
|---|---|---|---|---|
| Líneas de C | 8 | 7 | 14 | 7 |
| Tiempo en $[\mu s]$ | $3{,}4n^3$ | $13n^2$ | $46n\log n$ | $33n$ |
| Tiempo para $n = 10^2$ | $3{,}4\,[s]$ | $130\,[ms]$ | $30\,[ms]$ | $3{,}3\,[ms]$ |
| $10^3$ | $0{,}94\,[h]$ | $14\,[s]$ | $0{,}45\,[s]$ | $33\,[ms]$ |
| $10^4$ | 39 [dias] | 22 [min] | $6{,}1\,[s]$ | $0{,}33\,[s]$ |
| $10^5$ | 108 años | 1,5 días | 1,3 min | $3{,}3\,[s]$ |
| $10^6$ | 108 millones de años | 5 meses | 15 min | $33\,[s]$ |

Cuadro 1: Comparativa de Bentley [1] entre las variantes

Reportar la complejidad de un algoritmo en términos de $O(\cdot)$ es incompleto, pero el cuadro 1 muestra su relevancia. La ventaja es que la complejidad en estos términos es sencilla de obtener, en nuestros casos simples (algoritmos 1, 2, 3 y 5) por inspección, el teorema maestro da la complejidad para el algoritmo 4 directamente.

## Referencias

[1] Jon L. Bentley: *Algorithm design techniques.* Communications of the ACM, 27(9):865–871, September 1984.

```c
static double *aa;

static double msi(int l, int u)
{
  double Sum, MaxToRight, MaxToLeft, MaxCrossing,
         MaxInA, MaxInB;

  if(l >= u) /* Zero-element vector */
    return 0.0;
  if(l == u - 1) /* One-element vector */
    return max(aa[l], 0.0);

  int m = (l + u) / 2;
  /* Find max crossing to left */
  Sum = MaxToLeft = 0.0;
  for(int i = m - 1; i >= l; i--) {
    Sum += aa[i];
    MaxToLeft = max(MaxToLeft, Sum);
  }
  /* Find max crossing to right */
  Sum = MaxToRight = 0.0;
  for(int i = m; i < u; i++) {
    Sum += aa[i];
    MaxToRight = max(MaxToRight, Sum);
  }
  MaxCrossing = MaxToLeft + MaxToRight;

  MaxInA = msi(l, m);
  MaxInB = msi(m, u);

  return max(MaxCrossing, max(MaxInA, MaxInB));
}

double MaxSum(double a[], int n)
{
  aa = a;

  return msi(0, n);
}
```

Listado 4: Algoritmo 4: Usar sumas cumulativas

```c
double MaxSum(double a[], int n)
{
  double MaxSoFar, MaxEndingHere;

  MaxSoFar = MaxEndingHere = 0.0;
  for(int i = 0; i < n; i++) {
    MaxEndingHere = max(MaxEndingHere + a[i], 0.0);
    MaxSoFar = max(MaxSoFar, MaxEndingHere);
  }
  return MaxSoFar;
}
```

Listado 5: Algoritmo 5: Ir extendiendo resultado parcial