Estructuras de Datos
Clase 3 – Análisis de algoritmos recursivos

Dr. Sergio A. Gómez
http://cs.uns.edu.ar/~sag

Departamento de Ciencias e Ingeniería de la Computación
Universidad Nacional del Sur
Bahía Blanca, Argentina

Notación asintótica (Big-Oh)

Sean f(n) y g(n) : N → R

f(n) es O(g(n)) ssi existen c real con c>0 y n_0 natural con n_0>=1 tales que

$$f(n) \le cg(n) \text{ para todo } n \ge n_0$$

"f(n) es O(g(n))" se lee "f(n) es big-oh de g(n)" o "f(n) es del orden de g(n)"

Cálculo de tiempo de algoritmo recursivo

- Paso 1: Determinar la entrada
- Paso 2: Determinar el tamaño de la entrada
- Paso 3: Definir una recurrencia para T(n)
- Paso 4: Obtener una definición no recursiva para T(n)
- Paso 5: Determinar orden de tiempo de ejecución
- Paso 6: Hacer prueba por inducción para ver que las expresiones para T(n) de (3) y (4) son equivalentes.

Factorial

```
public static int fact( int n )
{
   if (n == 0)
        return 1;
   else
        return n *  fact(n-1);
}
```

Factorial

```
public static int fact( int n )
{
    if (n == 0)
        return 1;
    else
        return n *  fact(n-1);
}
```

En el caso base, testear n=0 junto con retornar 1 toma tiempo c_1, pues son dos operaciones primitivas.

En el caso recursivo, testear n=0, restar 1 a n, invocar fact (sin contar el tiempo que toma ejecutar fact(n-1), multiplicar por n y retornar toma tiempo c_2, pues son todas operaciones primitivas.

Factorial

- <u>Paso 1:</u> Entrada es “n”
- <u>Paso 2:</u> Tamaño de la entrada es “n”
- <u>Paso 3:</u> Definir recurrencia para T(n):

$$T(n) = \begin{cases} c_1 & si \quad n = 0 \\ c_2 + T(n-1) & si \quad n > 0 \end{cases}$$

Factorial

- Paso 4: Derivar definición no recursiva de T(n):

$T(n) = c_2 + T(n\text{-}1) = c_2 + (c_2 + T((n\text{-}1)\text{-}1)$

$= 2c_2 + T(n\text{-}2) = 2c_2 + (c_2 + T((n\text{-}2)\text{-}1)$

$= 3c_2 + T(n\text{-}3) = 3c_2 + (c_2 + T((n\text{-}3)\text{-}1)$

$= 4c_2 + T(n\text{-}4) = \ldots =$

$= ic_2 + T(n\text{-}i) \qquad (1)$

Termina cuando $n\text{-}i = 0$, luego $n = i$ (2)

Reemplazo (2) en (1) y obtengo:

$$T(n) = nc_2 + T(0) = nc_2 + c_1$$

- Paso 5: Obtener orden de tiempo de ejecución:

$$T(n) \text{ es } O(n)$$

Factorial

- <u>Paso 6:</u> Prueba por inducción de $T(n) = nc_2+c_1$

<u>Caso base:</u> $T(0) = c_1 = 0c_2 + c_1$

<u>Caso inductivo:</u>

$T(n) = c_2 + T(n-1) =$ *(x definición recursiva de T(n))*

$= c_2 + ((n-1)c_2 + c_1)$ *(x hipótesis inductiva)*

$= c_2 + (nc_2 - c_2 + c_1)$ *(x distributividad de *)*

$= c_2 + nc_2 - c_2 + c_1$ *(x asociatividad)*

$= nc_2 + c_1$ *(anulo c_2 positivo y negativo)*

Búsqueda binaria

<u>Problema:</u> Buscar entero “x” en arreglo de enteros ordenado “a” de “n” componentes

```
public static int bsearch( int [] a, int n, int x ) {
    return bsearch_aux( a, 0, n-1, x );
}
private static int bsearch_aux(int [] a, int ini, int fin, int x ) {
    if( ini <= fin ) {
        int medio = (ini + fin) / 2;
        if( a[medio] == x ) return medio;
        else if( a[medio] > x ) then
                return bsearch_aux( a, ini, medio-1, x)
            else
                return bsearch_aux( a, medio+1, fin , x)
    }
    else  return -1;  // x no está en el arreglo
}
```

- <u>Paso 1:</u> Entrada: Arreglo a y x
- <u>Paso 2:</u> Tamaño de entrada:

  n = cantidad de componentes de a
- <u>Paso 3:</u> Definición recursiva de T(n):

$$T(n) = \begin{cases} c_1 & , si \quad n = 0 \\ c_2 + T(\frac{n-1}{2}) & , si \quad n \geq 1 \end{cases}$$

- <u>Paso 4:</u> Obtener definición no recursiva de T(n)

$T(n) = c_2 + T((n-1)/2) = c_2 + (c_2 + T(((n-1)/2 - 1)/2)))$

$= 2c_2 + T((n-3)/4) = 2c_2 + (c_2 + T(((n-3)/4 - 1)/2)))$

$= 3c_2 + T((n-7)/8) = 3c_2 + (c_2 + T(((n-7)/8 - 1)/2)))$

$= 4c_2 + T((n-15)/16) = 4c_2 + (c_2 + T(((n-15)/16 - 1)/2)))$

$= 5c_2 + T((n-31)/32) = \ldots$

$= ic_2 + T((n-(2^i-1))/2^i)$ (1)

Termina cuando $(n-(2^i-1))/2^i = 0$, luego $n-(2^i-1) = 0$.

Entonces, $n-2^i+1=0$; por lo tanto, $n+1 = 2^i$ y $i = \log_2(n+1)$. (2)

Reemplazo (2) en (1):

$T(n) = \log_2(n+1)c_2 + T(0) = \log_2(n+1)c_2 + c_1$.

- Paso 5: Dar orden de tiempo de ejecución:

$$T(n) = \log_2(n+1)c_2 + c_1 \text{ es } O(\log_2(n+1)).$$

- Paso 6: Prueba inductiva (por inducción transfinita)

Caso base: $T(0) = c_1 = \log_2(0+1)c_2 + c_1 = c_1$

Caso inductivo: $T(n) = c_2 + T((n-1)/2)$ *(x def. T(n) rec.)*

$= c_2 + (\log_2((n-1)/2+1)c_2 + c_1)$ *(x hipótesis inductiva)*

$= c_2 + (\log_2((n-1+2)/2))c_2 + c_1$

$= c_2 + (\log_2(n+1) – \log_2(2))c_2 + c_1$

(xq log (a/b) = log(a) – log(b))

$= c_2 + (\log_2(n+1) – 1)c_2 + c_1$

$= c_2 + \log_2(n+1)c_2 - c_2+ c_1$

$= \log_2(n+1)c_2+ c_1.$

Merge sort

```
public static void mergesort( int [] a, int n)
{  msort( a, 0, n-1); }

private static void msort( int [] a, int ini, int fin)
{
    if( ini<fin) {
      int medio = (ini+fin)/2;
      msort(a, ini, medio );
      msort(a, medio+1,fin);
      merge( a, ini, medio, fin );  // hace un merge de los arreglos en O(n)
    }
}
```

```java
void merge( int [] a, int ini, int medio, int fin) {
  int i=ini, j=medio+1, k=0;
  int [] b = new int[fin-ini+1];
  while (i<=medio && j<=fin) {
      if (a[i] < a[j]) b[k++] = a[i++];
      else b[k++] = a[j++];
  }
  while (i<=medio) b[k++] = a[i++];
  while (j<=fin) b[k++] = a[j++];
  for(i=ini, k=0; i<=fin; i++, k++) a[i]=b[k];
}
```

T(n) = O(n) si n es el tamaño de arreglo a mezclar.

Tamaño de la entrada: n = cantidad de componentes de a

Recurrencia para n:

$$T(n) = \begin{cases} c_1 & si \quad n = 1 \\ c_2 n + 2T(n/2) & si \quad n > 1 \end{cases}$$

$$T(n) = c_2 n + 2T(n/2) = c_2 n + 2(c_2(n/2) + 2T(n/2/2))$$
$$= 2c_2 n + 4T(n/4) = 2c_2 n + 4(c_2(n/4) + 2T(n/4/2))$$
$$= 3c_2 n + 8T(n/8) = \ldots$$
$$= ic_2 n + 2^i T(n/2^i)$$

Termina cuando $n/2^i = 1$, es decir $n=2^i$.

Luego, $i = \log_2(n)$.

$T(n) = \log_2(n)c_2 n + nT(1) = \log_2(n)c_2 n + nc_1$ es $O(n\log_2(n))$