Divide y vencerás

Prof. Rhadamés Carmona

Última revisión: 30/03/2019

Agenda

- Definición
- Complejidad (caso recursivo)
- Exponenciación
- Búsqueda binaria
- Quick Sort, Merge Sort
- Mediana
- Torres de Hanoi
- Subárbol de máxima suma
- Multiplicación de matrices
- Tarea

Definición

- **Divide and conquer** es una técnica de diseño de algoritmos que consiste en descomponer casos complejos en casos más pequeños y fáciles de resolver, con un costo de combinar las soluciones obtenidas para construir la solución del problema original.
- Los problemas más pequeños se resuelven **independientemente** uno de otro. No hay overlapping entre subproblemas como sí occure en dynamic programming (DP).

Complejidad (caso recursivo)

- Como ya vimos en el tema de complejidad en tiempo, la complejidad de problemas recursivos que se suelen resolver con esta técnica tienen la siguiente función recurrente de tiempo:

  $T(1) = 1$ { se toma $T(1)=1$ por conveniencia }

  $T(n) = a.T(n/b)+f(n)$, si $n>1$

- $f(n)$ es el costo de fusionar las soluciones particionadas.

Complejidad (caso recursivo)

$T(1) = 1$

$T(n) = aT(n/b)+f(n)$, si $n>1$

$$a < f(b) \Rightarrow T(n) = \mathrm{O}(n^{\log_b^{f(n)}})$$

$$a = f(b) \Rightarrow T(n) = \mathrm{O}(n^{\log_b^a} . \log_b^n)$$

$$a > f(b) \Rightarrow T(n) = \mathrm{O}(n^{\log_b^a})$$

Exponenciación

- Cómo calcular X^n sin utilizar pow(x,n) ?
- Lo clásico son n-1 productos x*x*x*…*x, lo cual es cláramente O(n).
- La solución divide y conquista es inicialmente dividir el problema en 2 problemas de tamaño n/2, y luego combinar. Pero como cada problema de tamaño n/2 es “idéntico”, finalmente se reduce a un solo problema de tamaño n/2.

Exponenciación

$X^n = X^{n/2} * X^{n/2}$ si $n>0$ es par

$X^n = X * X^{n/2} * X^{n/2}$ si $n>0$ es impar

$X^n = 1$ si $n=0$

- Note que algorítmicamente, $X^{n/2}$ se calcula una vez (hacerlo 2 veces sería redundante), y que el costo de fusión es de apenas 1 o 2 multiplicaciones ($f(n)$=ctte=1).
- $a=1$, $f(n)=1$, $b=2$ ➔ $a = f(b) \Rightarrow T(n) = \mathrm{O}(n^{\log_b^a} . \log_b^n)$
- Claramente este algoritmo es O(log n).

Búsqueda binaria

- Se utiliza para buscar un elemento x "comparable" en un arreglo ordenado.
- Se revisa el elemento central del arreglo, y si no está allí, el problema se reduce a buscar en uno de los subarreglos izquierdo o derecho del elemento central, hasta encontrarlo o hasta que quede un solo elemento por revisar.
- Reduce un problema de tamaño n, a un problema de tamaño $n/2$, con un costo adicional $f(n)=1$.

Búsqueda binaria

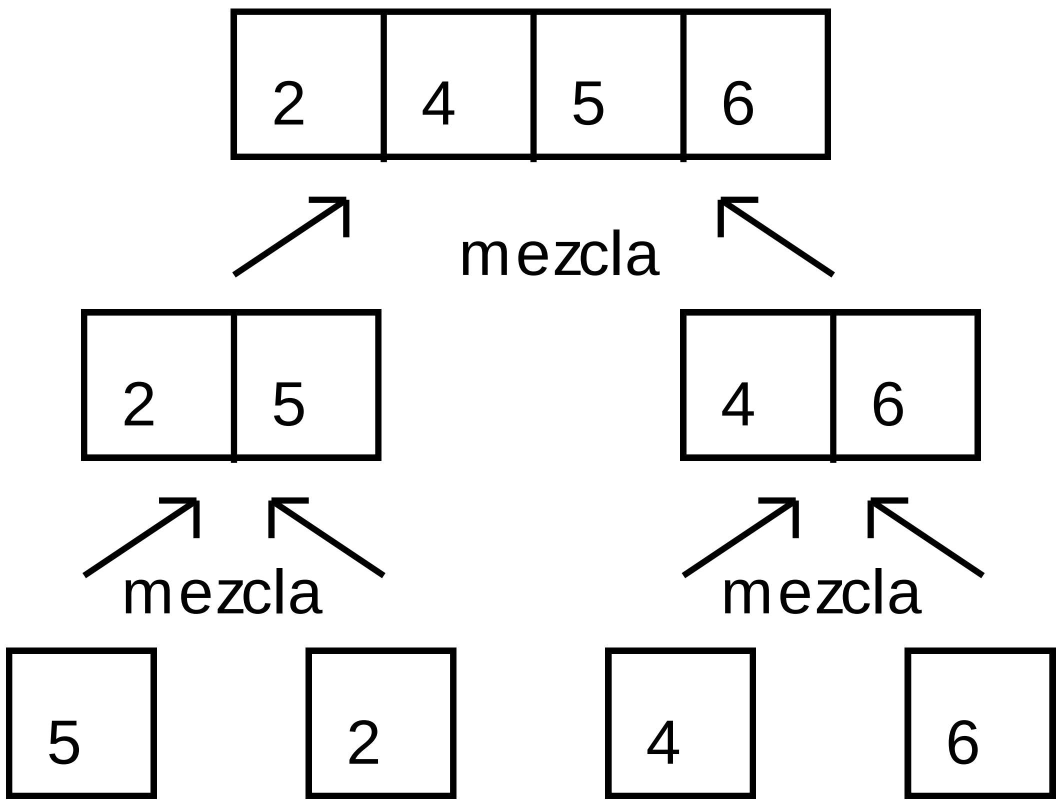
```
int binSearch(Key *A, const Key &x, int lo, int hi)
{
        if (hi < lo) return lo;
        int mid = lo + (hi - lo) / 2;
        if (A[mid] < x)
                return binSearch (A, x, lo, mid-1);
        if (A[mid] > x)
                return binSearch (A, x, mid+1, hi);
        return mid;
}
```

- Se reduce T(n) = 1*T(n/2)+1, por lo cual a=1, f(n)=1, b=2➔ $a = f(b) \Rightarrow T(n) = \mathrm{O}(n^{\log_b^a}.\log_b^n)$
- Claramente este algoritmo es O(log n).

Merge sort

- Consiste en dividir el arreglo en 2 partes
- Cada parte se divide recursivamente hasta llegar a un caso trivial (1 elemento)
- Al volver, se mezclan los arreglos ordenados
- Debido a que en el árbol de recursión de logn niveles siempre hay n elementos a mezclar (por nivel), el algoritmo es O(nlogn)
- $T(n)=2T(n/2)+n$

Merge sort



Merge sort

```
list MergeSort(list L)
{
        If  (L.size() <= 1)
                return_L;
        else
        {
                list L1 = Lista de los primeros n/2 elementos de L;
                list L2 = Lista de los últimos n/2 elementos de L;
                return (Mezcla(MergeSort(L1),MergeSort(L2)));
        }
}
```

$$T(n) = aT(n/b)+f(n) = 2T(n/2)+n$$

Es claro que merge sort divide un problema de tamaño n en dos sub problemas de tamaño n/2, con un costo de dividir y mezclar f(n)=n. Por lo tanto a=b=2, f(n)=n, a=f(b), quedando así

$$a = f(b) \Rightarrow T(n) = \mathrm{O}(n^{\log_b^a}.\log_b^n) \equiv \mathrm{O}(n.\log_2^n)$$

Quick sort

- Particiona el arreglo en 2 partes
- Luego ordena cada parte independientemente

```
void quicksort(elem a[], int l, int r)
{
        int (r<=l) return;
        int i = particion(a,l,r);
        quicksort(a, l, i-1);
        quicksort(a, i+1, r);
}
```

```
void particion(elem a[], int l, int r) {
        int i=l-1, j=r; elem v=a[r];
        for(;;)  {
                while(a[++i] < v);
                while (v < a[--j])
                        if (j==l) break;
                if (i>=j) break;
                swap(a[i],a[j]);
        }
        swap(a[i],a[r]);
        return i;
}
```

Quick sort

```
i           j,r
5 4 7 2 3 9 6
    i   j
5 4 3 2 7 9 6   swap1
      j i       break2
5 4 3 2 6 9 7   swap2
       (i)
```

Note que los menores quedan a la izquierda (l,…i-1) de la posición retornada (i) y los mayores a la derecha (i+1,…r)

```
void particion(elem a[], int l, int r) {
        int i=l-1, j=r; elem v=a[r];
        for(;;)  {
                while(a[++i] < v);
                while (a[--j] > v)
                    if (j==l) break;    //b1
                if (i>=j) break;        //b2
                swap(a[i],a[j]);        //s1
        }
        swap(a[i],a[r]);        //s2
        return i;
}
```

Quick sort

- El algoritmo de partición es claramente O(n) para una llamada de n elementos.
- El peor de los casos, es que al particionar k elementos resulte 2 conjuntos: uno de k-1 y otro de 1. Esto ocurre si el arreglo está ordenado.
- Si esto sucede en cada ocasión, en el peor caso se invoca partición N-1 veces, haciendo que el quick sort sea $O(n^2)$. $T(n)=T(n-1)+n$.
- La "esperanza" es que el pivote quede en el medio, en cuyo caso tenemos O(nlogn) como el merge sort. $T(n)=2T(n/2)+n$.
- En promedio (entre todas las posibles instancias), quick sort hace 39% más comparaciones que en el mejor de los casos.

Quick sort

- Cómo reducir la probabilidad de que ocurra el caso no deseado:
  - Desordene el arreglo antes de la invocación.
  - Tomar un elemento **aleatorio** como pivote, y no el más a la derecha.
  - Tomar los extremos y el centro (3 elementos), y seleccionar como pivote a la mediana entre estos tres. Esta es **la mejor opción**.
- Una aceleración posible, es usar insertion sort cuando el subarreglo sea pequeño (1-11 ?)

Mediana

- La mediana de n elementos es típicamente el elemento central después de ordenar los n elementos.
- Sin embargo, si utilizamos el proc. Partición, no es necesario ordenar los n elementos.
- Podemos llamar a partición recursivamente (solo en una rama del quick sot) hasta que la posición final del pivote caiga en la posición n/2.

Mediana

- Acorde a la posición retornada por partición (primera iteración), pueden suceder 3 casos:

  1) i=n/2: la mediana es a[i]
  2) i>n/2: el elemento buscado está en a[0..i-1]
  3) i>n/2: el elemento buscado está en a[i+1..n-1]

- Hay que re-escribir esta idea pensando ahora en los límites left y right del subarreglo remanente

Mediana

```
elem mediana(elem a[], int n)
{
        int left = 0, right = n-1;
        int pos = n/2;
        for (;;)
        {
                int i = particion(a,left,right);
                if (i==pos) return a[i];
                if (i>pos)  right=i-1;
                if (i<pos)  left =i+1;
        }
        return a[n/2]; // unreachable code!
}
```

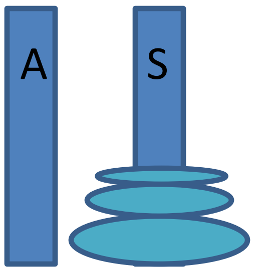
En vez de la mediana, podemos encontrar el k-ésimo elemento del ordenamiento (**algoritmo de selección**) sin necesidad de ordenar todo el arreglo, simplemente reemplazando "pos" por k. Este algoritmo en la práctica se comporta mejor que nlogn, típicamente lineal, pero se puede degenerar en $O(n^2)$.

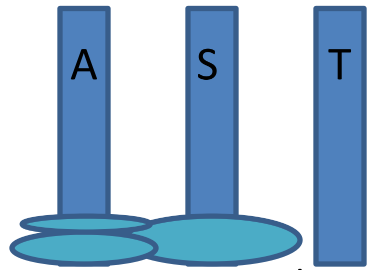
Torres de Hanoi

- 1883. Juego inventado por un matemático francés llamado Édouard Lucas.
- Hay 3 torres. La idea es mover la pila de discos de una torre hacia otra torre con las siguientes condiciones
  - Solo se mueve un disco a la vez, que esté en un tope
  - Un disco no puede colocarse sobre otro más pequeño
- La solución es recursiva. Mover N discos una posición "circular" a la derecha es mover N-1 discos a la izquierda, luego 1 disco a la derecha, y luego los N-1 discos moverlos otra posición circular más a la izquierda.

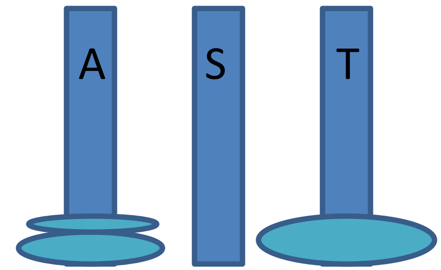
Torres de Hanoi

Viéndolo en el nivel más alto de recursión



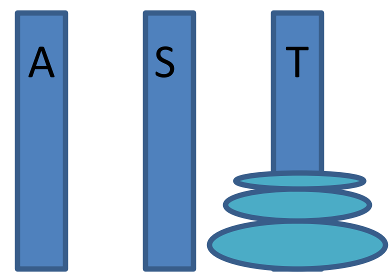


move(n - 1, source, auxiliary, target);



target.push(source.pop());

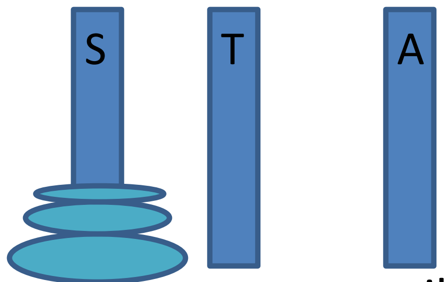
```
void move(n, source, target, auxiliary)
{
    if (n > 0) {
            move(n - 1, source, auxiliary, target);
            target.push(source.pop());
            move(n - 1, auxiliary, target, source);
    }
}
```



move(n - 1, auxiliary, target, source);

Torres de Hanoi

Paso a paso

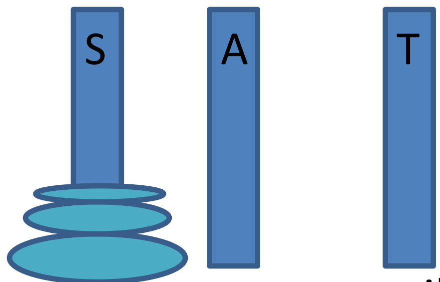


n=3

```
void move(n, source, target, auxiliary)
{
    if (n > 0) {
        move(n - 1, source, auxiliary, target);
        target.push(source.pop());
        move(n - 1, auxiliary, target, source);
    }
}
```

Torres de Hanoi

Paso a paso

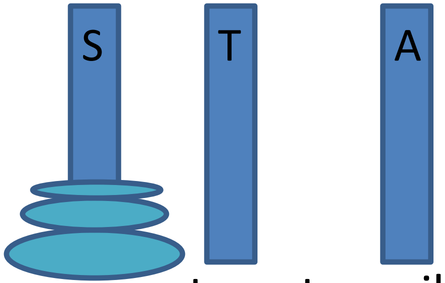


n=2

```
void move(n, source, target, auxiliary)
{
    if (n > 0) {
        move(n - 1, source, auxiliary, target);
        target.push(source.pop());
        move(n - 1, auxiliary, target, source);
    }
}
```

Torres de Hanoi

Paso a paso



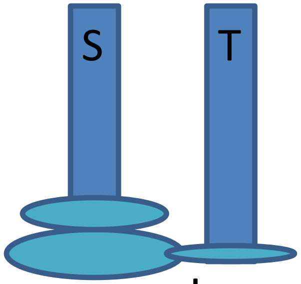
n=1

```
void move(n, source, target, auxiliary)
{
    if (n > 0) {
        move(n - 1, source, auxiliary, target);
        target.push(source.pop());
        move(n - 1, auxiliary, target, source);
    }
}
```

Sin efecto

Torres de Hanoi

Paso a paso

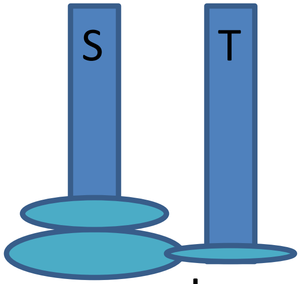


n=1

```
void move(n, source, target, auxiliary)
{
    if (n > 0) {
        move(n - 1, source, auxiliary, target);
        target.push(source.pop());
        move(n - 1, auxiliary, target, source);
    }
}
```

Torres de Hanoi

Paso a paso



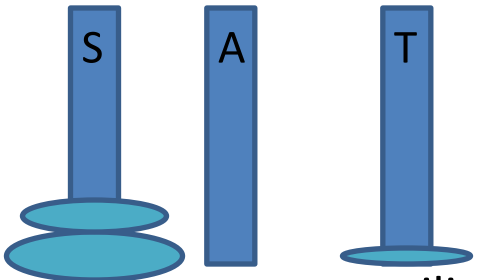
n=1

```
void move(n, source, target, auxiliary)
{
    if (n > 0) {
            move(n - 1, source, auxiliary, target);
            target.push(source.pop());
            move(n - 1, auxiliary, target, source);
    }
}
```

Sin efecto

Torres de Hanoi

Paso a paso



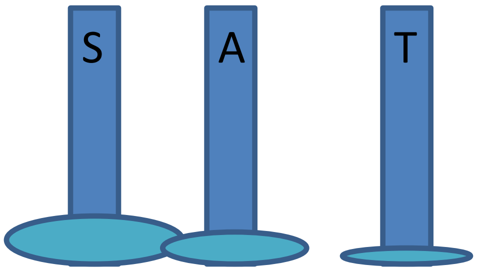
n=2

```
void move(n, source, target, auxiliary)
{
    if (n > 0) {
        move(n - 1, source, auxiliary, target);
        target.push(source.pop());
        move(n - 1, auxiliary, target, source);
    }
}
```

Ya ejecutado

Torres de Hanoi

Paso a paso

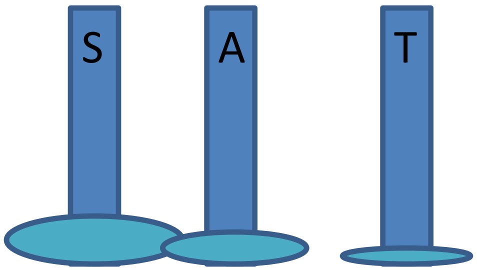


n=2

```
void move(n, source, target, auxiliary)
{
    if (n > 0) {
        move(n - 1, source, auxiliary, target);
        target.push(source.pop());
        move(n - 1, auxiliary, target, source);
    }
}
```

Torres de Hanoi

Paso a paso

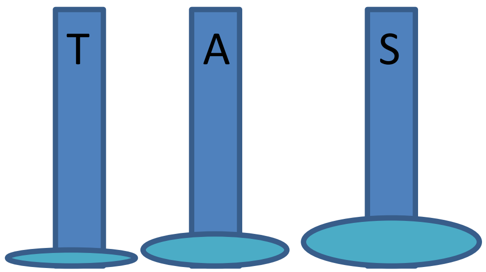


n=2

```
void move(n, source, target, auxiliary)
{
    if (n > 0) {
        move(n - 1, source, auxiliary, target);
        target.push(source.pop());
        move(n - 1, auxiliary, target, source);
    }
}
```

Torres de Hanoi

Paso a paso



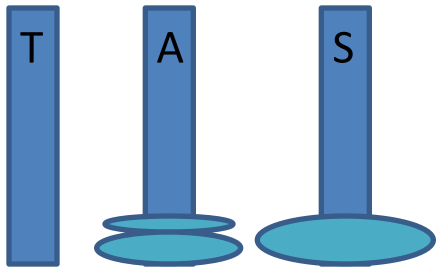
n=1

```
void move(n, source, target, auxiliary)
{
    if (n > 0) {
        move(n - 1, source, auxiliary, target);
        target.push(source.pop());
        move(n - 1, auxiliary, target, source);
    }
}
```

Sin efecto

Torres de Hanoi

Paso a paso

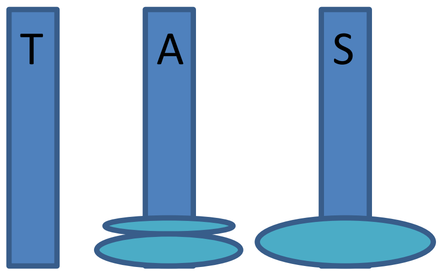


n=1

```
void move(n, source, target, auxiliary)
{
    if (n > 0) {
            move(n - 1, source, auxiliary, target);
            target.push(source.pop());
            move(n - 1, auxiliary, target, source);
    }
}
```

Torres de Hanoi

Paso a paso



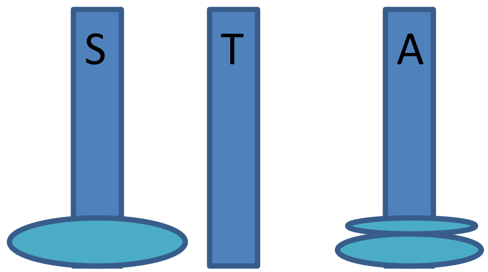
n=1

```
void move(n, source, target, auxiliary)
{
    if (n > 0) {
        move(n - 1, source, auxiliary, target);
        target.push(source.pop());
        move(n - 1, auxiliary, target, source);
    }
}
```

Sin efecto

Torres de Hanoi

Paso a paso



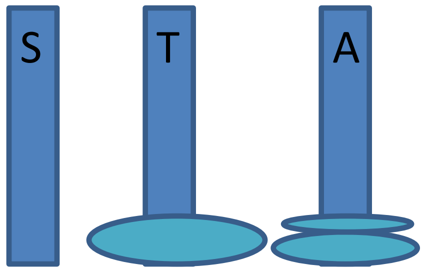
n=3

```
void move(n, source, target, auxiliary)
{
    if (n > 0) {
        move(n - 1, source, auxiliary, target);
        target.push(source.pop());
        move(n - 1, auxiliary, target, source);
    }
}
```

Ya ejecutado

Torres de Hanoi

Paso a paso

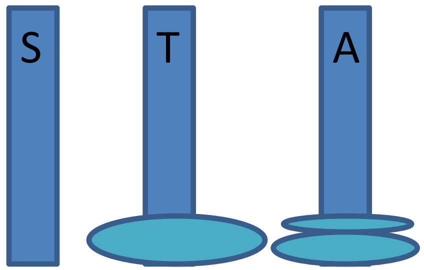


n=3

```
void move(n, source, target, auxiliary)
{
    if (n > 0) {
        move(n - 1, source, auxiliary, target);
        target.push(source.pop());
        move(n - 1, auxiliary, target, source);
    }
}
```

Torres de Hanoi

Paso a paso

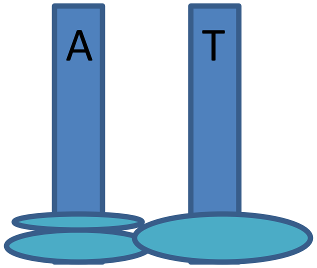


n=3

```
void move(n, source, target, auxiliary)
{
    if (n > 0) {
            move(n - 1, source, auxiliary, target);
            target.push(source.pop());
            move(n - 1, auxiliary, target, source);
    }
}
```

Torres de Hanoi

Paso a paso

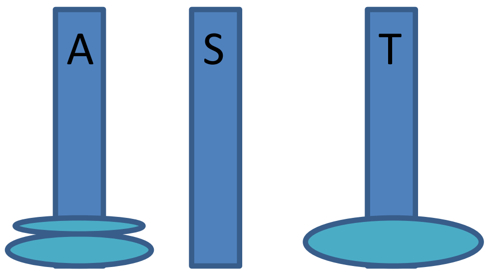


n=2

```
void move(n, source, target, auxiliary)
{
    if (n > 0) {
            move(n - 1, source, auxiliary, target);
            target.push(source.pop());
            move(n - 1, auxiliary, target, source);
    }
}
```

Torres de Hanoi

Paso a paso



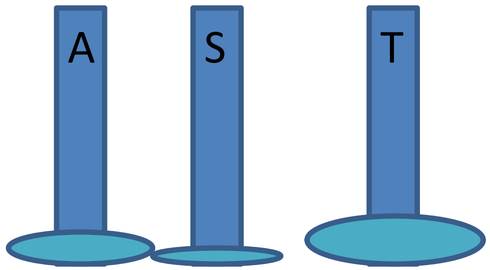
n=1

```
void move(n, source, target, auxiliary)
{
    if (n > 0) {
        move(n - 1, source, auxiliary, target);
        target.push(source.pop());
        move(n - 1, auxiliary, target, source);
    }
}
```

Sin efecto

Torres de Hanoi

Paso a paso

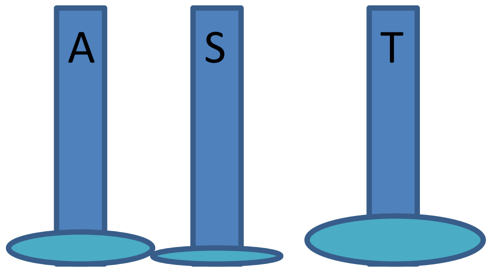


n=1

```
void move(n, source, target, auxiliary)
{
    if (n > 0) {
        move(n - 1, source, auxiliary, target);
        target.push(source.pop());
        move(n - 1, auxiliary, target, source);
    }
}
```

Torres de Hanoi

Paso a paso



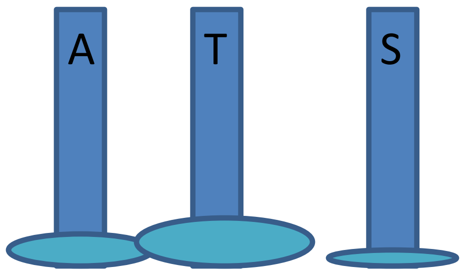
n=1

```
void move(n, source, target, auxiliary)
{
    if (n > 0) {
        move(n - 1, source, auxiliary, target);
        target.push(source.pop());
        move(n - 1, auxiliary, target, source);
    }
}
```

Sin efecto

Torres de Hanoi

Paso a paso



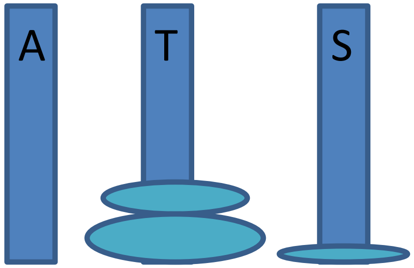
n=2

```
void move(n, source, target, auxiliary)
{
    if (n > 0) {
        move(n - 1, source, auxiliary, target);
        target.push(source.pop());
        move(n - 1, auxiliary, target, source);
    }
}
```

Ya ejecutado

Torres de Hanoi

Paso a paso

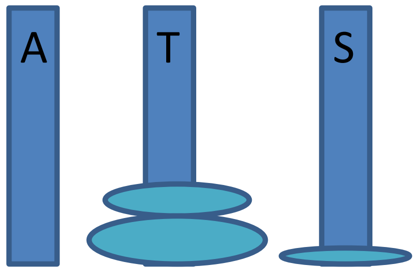


n=2

```
void move(n, source, target, auxiliary)
{
    if (n > 0) {
        move(n - 1, source, auxiliary, target);
        target.push(source.pop());
        move(n - 1, auxiliary, target, source);
    }
}
```

Torres de Hanoi

Paso a paso

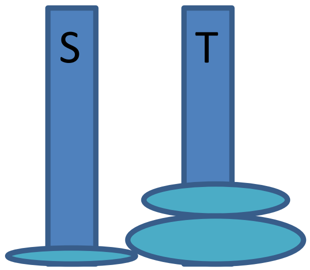


n=2

```
void move(n, source, target, auxiliary)
{
    if (n > 0) {
            move(n - 1, source, auxiliary, target);
            target.push(source.pop());
            move(n - 1, auxiliary, target, source);
    }
}
```

Torres de Hanoi

Paso a paso



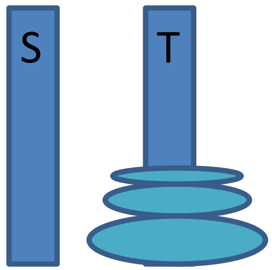
n=1

```
void move(n, source, target, auxiliary)
{
    if (n > 0) {
        move(n - 1, source, auxiliary, target);
        target.push(source.pop());
        move(n - 1, auxiliary, target, source);
    }
}
```

Sin efecto

Torres de Hanoi

Paso a paso

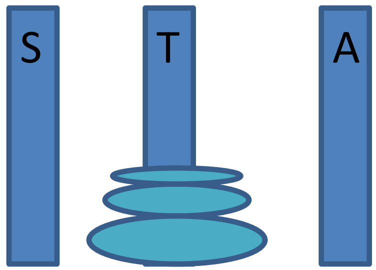


n=1

```
void move(n, source, target, auxiliary)
{
    if (n > 0) {
        move(n - 1, source, auxiliary, target);
        target.push(source.pop());
        move(n - 1, auxiliary, target, source);
    }
}
```

Torres de Hanoi

Paso a paso



n=1

```
void move(n, source, target, auxiliary)
{
    if (n > 0) {
        move(n - 1, source, auxiliary, target);
        target.push(source.pop());
        move(n - 1, auxiliary, target, source);
    }
}
```

Sin efecto

Torres de Hanoi

```
void move(n, source, target, auxiliary)
{
    if (n > 0) {
            move(n - 1, source, auxiliary, target);
            target.push(source.pop());
            move(n - 1, auxiliary, target, source);
    }
}
```

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

Torres de Hanoi

$T(n) = 2T(n-1) + 1 = 2^n-1$?

Por inducción, $T(0) = 0$, $T(1)=2*T(0)+1=1=2^1-1$

Supongamos cierto para n-1

$T(n-1) = 2^{n-1}-1$

Demostremos ahora la tesis usando la hip.

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

$\Rightarrow T(n) = 2\ (2^{n-1}-1) + 1$

$\Rightarrow T(n) = 2^n-2 + 1$

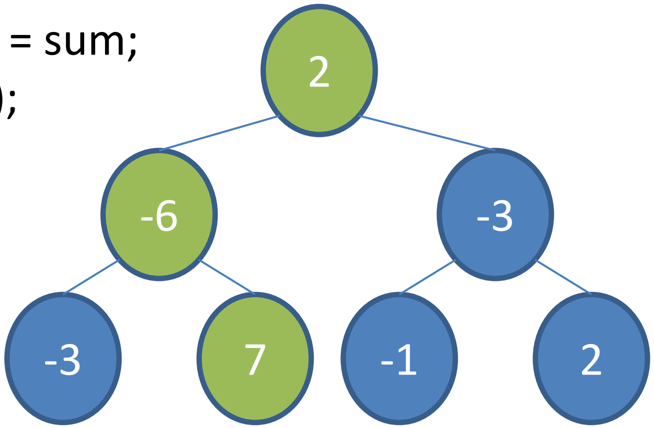
$\Rightarrow T(n) = 2^n-1 = O(2^n)$

Subárbol de máxima suma

- Dado un árbol binario, determinar el subárbol cuya suma de los elementos sea máxima, pero solo imprimir dicha suma.
- Si todos los elementos son positivos, la solución es trivial (el árbol mismo).
- El problema es la existencia de negativos.

Subárbol de máxima suma

```
int f(Node* root, int& maxsum)
{
        if (root == NULL)
                return 0;
        int sum = root->value+  f(root->left, max) + f(root->right, max);
        if (sum > maxsum)
                maxsum = sum;
        return max(0,sum);
}
```



Esta solución es O(n), y se recorre el árbol en post-orden. Cómo obtener luego el subárbol solución?

Multiplicación de matrices

- Calcular, C = A*B, $O(n^3)$

```
for (int i=0; i<n; i++)
    for (int j=0; j<n; j++)
    {
        c[i][j] = 0;
        for (int k=0; k<n; k++)
            c[i][j] += a[i][k] * b[k][j];

    }
```

Multiplicación de matrices

- Calcular, C = A*B, $O(n^3)$, versión divide y conquista

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_{1,1} & \mathbf{A}_{1,2} \\ \mathbf{A}_{2,1} & \mathbf{A}_{2,2} \end{bmatrix}, \mathbf{B} = \begin{bmatrix} \mathbf{B}_{1,1} & \mathbf{B}_{1,2} \\ \mathbf{B}_{2,1} & \mathbf{B}_{2,2} \end{bmatrix}, \mathbf{C} = \begin{bmatrix} \mathbf{C}_{1,1} & \mathbf{C}_{1,2} \\ \mathbf{C}_{2,1} & \mathbf{C}_{2,2} \end{bmatrix}$$

$$\begin{aligned}
\mathbf{C}_{1,1} &= \mathbf{A}_{1,1}\mathbf{B}_{1,1} + \mathbf{A}_{1,2}\mathbf{B}_{2,1} \\
\mathbf{C}_{1,2} &= \mathbf{A}_{1,1}\mathbf{B}_{1,2} + \mathbf{A}_{1,2}\mathbf{B}_{2,2} \\
\mathbf{C}_{2,1} &= \mathbf{A}_{2,1}\mathbf{B}_{1,1} + \mathbf{A}_{2,2}\mathbf{B}_{2,1} \\
\mathbf{C}_{2,2} &= \mathbf{A}_{2,1}\mathbf{B}_{1,2} + \mathbf{A}_{2,2}\mathbf{B}_{2,2}
\end{aligned}$$

Note que aún requiere $n^3 = 8$ multiplicaciones, siendo un approach divide y conquista, en donde en cada multiplicación podemos aplicar el mismo principio.

Multiplicación de matrices

- Calcular, C = A*B, usando Strassen algorithm (finales de los 60's), $O(n^{2.8074})$

M1 = (A12 – A22).(B21 + B22)
M2 = (A11 + A22).(B11 + B22)
M3 = (A11 – A21).(B11 + B21)
M4 = (A11 + A12).B22
M5 = A11 .(B12 –B22)
M6 = A22.(B21 – B11)
M7 = (A21 + A22).B11

$$\mathbf{C}_{1,1} = \mathbf{M}_1 + \mathbf{M}_4 - \mathbf{M}_5 + \mathbf{M}_7$$
$$\mathbf{C}_{1,2} = \mathbf{M}_3 + \mathbf{M}_5$$
$$\mathbf{C}_{2,1} = \mathbf{M}_2 + \mathbf{M}_4$$
$$\mathbf{C}_{2,2} = \mathbf{M}_1 - \mathbf{M}_2 + \mathbf{M}_3 + \mathbf{M}_6$$

Note que hay 7 multiplicaciones (y no 8), pero muchas sumas "cuadráticas". Sin embargo, hay una ganancia en número de operaciones que se logra observar para n grande.

Multiplicación de matrices

- Actualmente el algoritmo de Coppersmith-Winograd (2008) permite multiplicar matrices en O(N^2.376).

Tarea

- Implementar el algoritmo de strassen y comparar contra el método cúbico para n=16, 32, 64,128, 256, 512. Arrojar los tiempos promedio en cada caso, e indicar el % de diferencia en tiempo. Enviar código fuente, resultados obtenidos y una discusión de 1 párrafo sobre estos resultados.
- Implementar quick sort, tomando como pivote a la mediana entre el centro y los 3 extremos. Comparar merge sort con el quick sort original, y este quick sort mejorado, para arreglos de distintos tamaños (selecciones 20 tamaños entre 1 y 100 millones), considerando 3 casos: arreglos desordenados, arreglos ordenados asc. y ordenados des. Imprimir el tiempo promedio en cada caso, por cada tamaño de arreglo. Enviar código fuente, resultados obtenidos y una discusión de 1 párrafo sobre estos resultados.
- Buscar 3 problemas que se resuelven con “divide y conquista”, explicar y codificar su solución.

Bibliografía recomendada

- Robert Sedgewick. “Algorithms in C++”, Third edition, parts 1-4. Addison-Wesley, 1998.