



Diseño y Análisis de Algoritmos

Diseño de Algoritmos

Dr. Jesús Ariel Carrasco Ochoa
ariel@inaoep.mx
Oficina 8311



Contenido

- Algoritmos tipo Divide y Vencerás
- Algoritmos Ávidos
- Programación Dinámica



Algoritmos tipo Divide y Vencerás

Idea

- Dividir la instancia del problema a resolver en subinstancias de menor tamaño
- Solucionar cada una de las subinstancias
- Combinar las soluciones para obtener la solución de la instancia original

Algoritmos tipo Divide y Vencerás

Estructura General DC(X)

- Si X es suficientemente pequeño regresar Solución(X)
- Descomponer X en subinstancias $X_1, X_2, \dots, X_c$
- Para $i=1, \dots, c$: $Y_i = DC(X_i)$
- Combinar Y_i ; $i=1, \dots, c$ para obtener Y la solución de X
- Regresar Y

Algoritmos tipo Divide y Vencerás

Análisis de DC(X)

- Sea n/b el tamaño aproximado de cada una de c subinstancias de X (de tamaño n), para algún b entero
- Sea $g(b)$ el tiempo requerido para dividir un problema de tamaño n en c subinstancias y después combinar las soluciones
- Sea $h(n)$ el tiempo que toma resolver una instancia pequeña de tamaño $n \leq n_0$

Algoritmos tipo Divide y Vencerás

Análisis de DC(X)

- Como los algoritmos tipo divide y vencerás son recursivos el tiempo se plantea como una recurrencia

$$T(1) = h(n) \quad \text{para } n \leq n_0$$

$$T(n) = cT\left(\left\lfloor \frac{n}{b} \right\rfloor\right) + g(n)$$

Algoritmos tipo Divide y Vencerás

Si $g(n) = \Theta(n^k)$

Se aplica el método maestro:

$$T(n) = \begin{cases} \Theta(n^k) & \text{si } c < b^k \\ \Theta(n^k \log_b(n)) & \text{si } c = b^k \\ \Theta(n^{\log_b c}) & \text{si } c > b^k \end{cases}$$



Ejemplo: Búsqueda Binaria

Dado un arreglo ordenado de números decidir si un cierto número x está o no en el arreglo



Ejemplo: Búsqueda Binaria

Dado un arreglo ordenado de números decidir si un cierto número x está o no en el arreglo

Dado que el arreglo está ordenado se puede proceder de la siguiente manera:

Si el arreglo tiene un solo elemento, éste se compara con x y se responde si es igual o no

Comparar x con el número ubicado a la mitad del arreglo y si:

- x es igual responder "Sí"
- x es menor buscarlo en la primera mitad
- x es mayor buscarlo en la segunda mitad

Ejemplo: Búsqueda Binaria (BB)

Entrada: a un arreglo ordenado de números
ini,fin inicio y fin del arreglo
x el número a buscar

Salida: Sí o No

```
If( ini == fin ) if( x == a[ini] ) return "Sí"
                  else return "No"
```

```
mitad = (ini+fin)/2
```

```
if( x == a[mitad] ) return "Sí"
```

```
else
```

```
    if( x < a[mitad] ) return BB(a,ini,mitad-1,x)
```

```
        else return BB(a,mitad+1,fin,x)
```

Ejemplo: Búsqueda Binaria

En este caso: $h(n) = \Theta(1)$
 $g(n) = \Theta(1)$

Con lo que:

$c=1$ porque solo se debe solucionar uno de los subproblemas

$b=2$ porque cada subproblema es de tamaño $n/2$ (aprox) con $n = \text{fin} - \text{ini} + 1$

$k=0$ porque $n^0 = 1$

Por lo tanto $c = b^k$ y entonces:

$$T(n) = \Theta(n^0 \log(n)) = \Theta(\log(n))$$



Ejemplo: Ordenamiento por mezclas

Dado un arreglo A de números, para ordenarlo se puede proceder de la siguiente manera:

- Separar A en dos mitades A1 y A2
- Ordenar A1 y A2
- Mezclar A1 y A2 ya ordenados para obtener el arreglo completo ordenado

Ejemplo: Ordenamiento por mezclas mergeSort

Entrada: a un arreglo de números
n el tamaño del arreglo

Salida: a ordenado
Para todo $i=1, \dots, n-1$ $a[i] \leq a[i+1]$

```
If( n == 1 ) regresar
k = n/2
for( i=0 ; i<k ; i++) a1[i] = a[i]
for( i=k ; i<n ; i++) a2[i-k] = a[i]
mergeSort(a1,k)
mergeSort(a2,n-k)
mezclar(a1,k,a2,n-k,a)
```

Ejemplo: mezclar 2 arreglos ordenados

Entrada: a_1, a_2 los arreglos a mezclar
 n, m los tamaños de los arreglos

Salida: a ordenado
Para todo $i=1, \dots, n-1$ $a[i] \leq a[i+1]$

$i=0$

$j=0$

$k=0$

while($i < n$ && $j < m$)

 If($a_1[i] < a_2[j]$) { $a[k]=a_1[i]; i++; k++$ }

 else { $a[k]=a_2[j]; j++; k++$ }

while($i < n$) { $a[k]=a_1[i]; i++; k++$ }

while($j < m$) { $a[k]=a_2[j]; j++; k++$ }

Ejemplo: Ordenamiento por mezclas

En este caso: $h(n) = \Theta(1)$
 $g(n) = \Theta(n)$

Con lo que:

$c=2$ porque se deben solucionar dos subproblemas

$b=2$ porque cada subproblema es de tamaño $n/2$ (aprox)

$k=1$

Por lo tanto $c=b^k$ y entonces:

$$T(n) = \Theta(n \log(n))$$



Ejemplo: QuickSort

Dado un arreglo A de números, para ordenarlo se puede proceder de la siguiente manera:

- Seleccionar uno de los elementos de A
- Ubicarlo en su posición correcta, para esto pasar de un lado los elementos mayores y del otro lado los menores o iguales
- Ordenar cada uno de los lados

Ejemplo: QuickSort

Entrada: a un arreglo de números
ini, fin índices inicial y final de a

Salida: a ordenado
Para todo $i=1, \dots, N-1$ $a[i] \leq a[i+1]$

```
izq = ini;  
der = fin;  
pivote = a[(izq+der)/2];
```

Ejemplo: QuickSort

Repeat

```
while( a[izq]<pivote && izq<fin )izq++;  
while( pivote<a[der] && der>ini )der--;  
if(izq <= der)  
{  
    temporal = a[izq];  
    a[izq] = a[der];  
    a[der] = temporal;  
    izq++;  
    der--;  
}  
until (izq > der);
```

Ejemplo: QuickSort

```
if( ini < der )  
    qs( a, ini ,der );
```

```
if( fin > izq )  
    qs( a, izq, fin);
```

Ejemplo: QuickSort

Si el pivote queda siempre aproximadamente a la mitad entonces:

$$h(n) = \Theta(1)$$

$$g(n) = \Theta(n)$$

Con lo que:

$c=2$ porque se deben solucionar dos subproblemas

$b=2$ porque cada subproblema es de tamaño $n/2$

$k=1$

Por lo tanto $c=b^k$ y entonces:

$$T(n) = \Theta(n \log(n))$$



Ejemplo: QuickSort

Sin embargo, si el pivote queda siempre en uno de los extremos entonces:

- Quicksort se llama recursivamente n veces
- Y cada llamado es $\Theta(n)$

Por lo tanto en este caso (que es el peor caso) QuickSort es $\Theta(n^2)$



QuickSort vs. MergeSort

Por qué si MergeSort es siempre $\Theta(n \log(n))$ y QuickSort es en el peor caso $\Theta(n^2)$ se usa más QuickSort que MergeSort ?



QuickSort vs. MergeSort

Por qué si MergeSort es siempre $\Theta(n \log(n))$ y QuickSort es en el peor caso $\Theta(n^2)$ se usa más QuickSort que MergeSort ?

Debido al almacenamiento ?



QuickSort vs. MergeSort

Por qué si MergeSort es siempre $\Theta(n \log(n))$ y QuickSort es en el peor caso $\Theta(n^2)$ se usa más QuickSort que MergeSort ?

Debido al almacenamiento ?

En ambos casos el espacio requerido es $\Theta(n)$



QuickSort vs. MergeSort

Por qué si MergeSort es siempre $\Theta(n \log(n))$ y QuickSort es en el peor caso $\Theta(n^2)$ se usa más QuickSort que MergeSort ?

Debido al almacenamiento ?

En ambos casos el espacio requerido es $\Theta(n)$

Sin embargo, MergeSort requiere un espacio de tamaño $2n$ y Quicksort de tamaño n y cuando se quieren ordenar muchos elementos esto es muy importante



Algoritmos Ávidos, Voraces, Greedy

Comúnmente usados para optimización

Optan por lo mejor en cada momento y nunca reconsideran

Comúnmente obtienen soluciones buenas pero subóptimas

Algoritmos Ávidos, Voraces, Greedy

La forma general es la siguiente

```
S = ∅  
while( (C != ∅) && !solucion(S) ) {  
    x = selecciona(C)  
    C = C - {x}  
    if( factible(S ∪ {x}) ) S = S ∪ {x}  
}  
if( solucion(S) ) return S  
    else return ∅
```



Algoritmos Ávidos, Voraces, Greedy

El análisis se hace dependiendo de cada caso, pero en general depende de:

- Costo de decidir si se alcanzó una solución
- Costo de seleccionar el mejor siguiente paso
- Costo de decidir si es factible alcanzar la solución después de un cierto paso
- Número de repeticiones del ciclo de selección



Ejemplo: Entregar una cierta cantidad de dinero

Se tienen billetes y/o monedas de ciertas denominaciones

Se quiere entregar una cierta cantidad de dinero utilizando la menor cantidad de billetes y/o monedas



Ejemplo: Entregar una cierta cantidad de dinero

Un posible algoritmo ávido procedería como sigue:

1. Seleccionar el billete o moneda de mayor valor que no sobrepase al cantidad a entregar
2. Entregar el billete o moneda seleccionado
3. Restar el valor del billete o moneda seleccionado de la cantidad a entregar
4. Si lo que falta por entregar no es 0 repetir desde el inicio

Ejemplo: Entregar una cierta cantidad de dinero

Entrada: S la cantidad a entregar
C el conjunto de billetes o monedas disponibles antes de entregar S

Salida: E Los billetes y monedas a entregar o $\emptyset$
C el conjunto de billetes o monedas disponibles despues de entregar S

E = $\emptyset$; F = S

```
while( (C !=  $\emptyset$ ) && (F == 0) ) {  
    x = selecciona(C)  
    C = C - {x}; E = E  $\cup$  {x}  
    if( !factible(F,C) ) break;  
    else F = F - X  
}  
if( F == 0 ) return E,C  
    else { C = C  $\cup$  E; return  $\emptyset$ ,C }
```

Ejemplo: Entregar una cierta cantidad de dinero

En este caso:

- Costo de decidir si se alcanzó una solución es $O(1)$
- Costo de seleccionar el mejor siguiente paso es $O(1)$
- Costo de decidir si es factible alcanzar la solución después de un cierto paso es $O(1)$
- Número de repeticiones del ciclo de selección es $O(S)$

Por lo tanto este algoritmo es $O(S)$



Programación Dinámica

En esencia consiste en evitar calcular lo mismo más de una vez y para esto se almacena lo ya calculado

Es útil para optimizar algoritmos recursivos en los cuales se presenta el problema de superposición, es decir una solución parcial se utiliza más de una vez para resolver instancias mayores.

Ejemplo: Números de Fibonacci

Entrada: `n` el número a generar

Salida: `fn` el `n`-ésimo número de Fibonacci

```
If( n ≤ 2 )
```

```
    fn = 1;
```

```
else
```

```
    fn = fib(n-1)+fib(n-2)
```

Como vimos este algoritmo es ineficiente y esto se debe a que algunos cálculos se repiten varias veces

Ejemplo: Números de Fibonacci

Utilizando programación dinámica este algoritmo se puede mejorar como:

Entrada: `n` el número a generar

Salida: `fn` el `n`-ésimo número de Fibonacci

```
If( n ≤ 2 )  
    fn = 1;  
else  
    if( f[n] == -1 ) f[n] = fib(n-1)+fib(n-2)  
    return f[n]
```

Este algoritmo es $O(n)$



Diseño y Análisis de Algoritmos

Diseño de Algoritmos

Dr. Jesús Ariel Carrasco Ochoa
ariel@inaoep.mx
Oficina 8311