Algoritmos y Estructuras de Datos II

Recurrencias Divide y Vencerás
Jerarquía de Funciones

25 y 27 de marzo de 2019

Contenidos

Recurrencias

- Surgen al analizar algoritmos recursivos, como la ordenación por intercalación.
- El conteo de operaciones "copia" la recursión del algoritmo y se vuelve recursivo también.
- Ejemplo: máximo de comparaciones de la ordenación por intercalación.
- Es un ejemplo de algoritmo divide y vencerás.
- Es un ejemplo de recurrencia divide y vencerás:

$$t(n) = \begin{cases} 0 & \text{si } n \in \{0, 1\} \\ t(\lceil n/2 \rceil) + t(\lfloor n/2 \rfloor) + n - 1 & \text{si } n > 1 \end{cases}$$

Algoritmo divide y vencerás

Características

- hay una solución para los casos sencillos,
- para los complejos, se **divide** o **descompone** el problema en subproblemas:
  - cada subproblema es de igual naturaleza que el original,
  - el tamaño del subproblema es una **fracción** del original,
  - se resuelven los subproblemas apelando al mismo algoritmo,
- se **combinan** esas soluciones para obtener una solución del original.

Algoritmo divide y vencerás
Forma general

fun DyV(x) **ret** y
  if x suficientemente pequeño o simple **then** y:= ad_hoc(x)
  else descomponer x en $x_1, x_2, \ldots, x_a$
    for i:= 1 **to** a **do** y_i:= DyV(x_i) **od**
    combinar $y_1, y_2, \ldots, y_a$ para obtener la solución y de x
  fi
end fun

Normalmente los x_i son **fracciones** de x:

$$|x_i| = \frac{|x|}{b}$$

para algún b fijo mayor que 1.

Algoritmo divide y vencerás
Ejemplos

- Ordenación por intercalación:
  - "x simple" = fragmento de arreglo de longitud 0 ó 1
  - "descomponer" = partir al medio ($b = 2$)
  - $a = 2$
  - "combinar" = intercalar
- Ordenación rápida:
  - "x simple" = fragmento de arreglo de longitud 0 ó 1
  - "descomponer" = separar los menores de los mayores ($b = 2$)
  - $a = 2$
  - "combinar" = yuxtaponer

Forma general de DyV en python

```
def DyV(x):
    if simple(x):
        y = ad_hoc(x)
    else:
        xs = decompose(x)
        ys = [DyV(xi) for xi in xs]
        y = combine(ys)
    return y
```

msort usando DyV en python

Caso base y llamada principal

```
def simple(x):
    lft,rgt = x[1],x[2]
    return lft == rgt

def ad_hoc(x):
    return x

def msort(a):
    z = DyV((a,0,len(a)-1))
    return z[0]

h = msort([5,2,9,7])
```

msort usando DyV en python

Caso recursivo

```
def decompose(x):
    c,lft,rgt= x[0],x[1],x[2]
    mid = (lft+rgt) // 2
    xs = [(c,lft,mid),(c,mid+1,rgt)]
    return xs

def combine(ys):
    c,lft,mid,rgt = ys[0][0],ys[0][1],ys[0][2],ys[1][2]
    merge(c,lft,mid,rgt)
    return c,lft,rgt
```

qsort usando DyV en python

Caso base y llamada principal

```
def simple(x):
    lft,rgt = x[1],x[2]
    return lft >= rgt

def ad_hoc(x):
    return x

def qsort(a):
    z = DyV((a,0,len(a)-1))
    return z[0]

h = qsort([5,2,9,7])
```

qsort usando DyV en python

Caso recursivo

```
def decompose(x):
    c,lft,rgt= x[0],x[1],x[2]
    ppiv = partition(c,lft,rgt)
    xs = [(c,lft,ppiv-1),(c,ppiv+1,rgt)]
    return xs

def combine(ys):
    c,lft,rgt = ys[0][0],ys[0][1],ys[1][2]
    return c,lft,rgt
```

Algoritmo divide y vencerás
Forma general

fun DyV(x) **ret** y
    if x suficientemente pequeño o simple **then** y:= ad_hoc(x)
    else descomponer x en $x_1, x_2, \ldots, x_a$
        for i:= 1 **to** a **do** y_i:= DyV(x_i) **od**
        combinar $y_1, y_2, \ldots, y_a$ para obtener la solución y de x
    fi
end fun

- a: número de llamadas recursivas a DyV.
- b: relación entre el tamaño de x y el de x_i, satisface $|x_i| = \frac{|x|}{b}$.
- k: el orden de descomponer y combinar es n^k.

Algoritmo divide y vencerás

Conteo

Si queremos contar el costo computacional (número de operaciones) $t(n)$ de la función *DyV* obtenemos:

$$t(n) = \begin{cases} c & \text{si la entrada es pequeña o simple} \\ a * t(n/b) + g(n) & \text{en caso contrario} \end{cases}$$

si c es una constante que representa el costo computacional de la función ad_hoc y $g(n)$ es el costo computacional de los procesos de descomposición y de combinación.
Esta definición de $t(n)$ es recursiva (como el algoritmo *DyV*), se llama **recurrencia**. Existen distintos tipos de recurrencia. Ésta se llama **recurrencia divide y vencerás**.

Recurrencias divide y vencerás

Por la forma de la recurrencia

$$t(n) = \begin{cases} c & \text{si la entrada es pequeña o simple} \\ a * t(n/b) + g(n) & \text{en caso contrario} \end{cases}$$

resulta más sencillo calcular $t(n)$ cuando n es potencia de b.
Se organiza la tarea en dos partes

- calcular el orden de $t(n)$ cuando n es potencia de b,
- extender el cálculo para los demás n.

Calculando para n potencia de b

- Supongamos que
  - $$t(n) = \begin{cases} c & \text{si la entrada es pequeña o simple} \\ a * t(n/b) + g(n) & \text{en caso contrario} \end{cases}$$
  - y $g(n)$ es del orden de n^k, es decir $g(n) \leq dn^k$ para casi todo $n \in \mathbb{N}$
- n potencia de b, $n = b^m$
- $$\begin{aligned} t(n) &= t(b^m) \\ &= a * t(b^m/b) + g(b^m) \\ &\leq a * t(b^{m-1}) + d(b^m)^k \\ &\leq a * t(b^{m-1}) + d(b^k)^m \end{aligned}$$

Iterando

$$
\begin{aligned}
t(b^m) &\leq at(b^{m-1}) + d(b^k)^m \\
&\leq a(at(b^{m-2}) + d(b^k)^{m-1}) + d(b^k)^m \\
&\leq a^2 t(b^{m-2}) + ad(b^k)^{m-1} + d(b^k)^m \\
&\leq a^3 t(b^{m-3}) + a^2 d(b^k)^{m-2} + ad(b^k)^{m-1} + d(b^k)^m \\
&\leq \ldots \\
&\leq a^m t(1) + a^{m-1} db^k + \ldots + ad(b^k)^{m-1} + d(b^k)^m \\
&= a^m c + d(b^k)^m((a/b^k)^{m-1} + \ldots + a/b^k + 1) \\
&= a^m c + d(b^m)^k(r^{m-1} + \ldots + r + 1)
\end{aligned}
$$

donde $r = a/b^k$ y recordemos $n = b^m$ y entonces $m = \log_b n$

$$
\begin{aligned}
t(n) &\leq a^{\log_b n} c + dn^k(r^{m-1} + \ldots + r + 1) \\
&= n^{\log_b a} c + dn^k(r^{m-1} + \ldots + r + 1)
\end{aligned}
$$

Propiedad del logaritmo

En el último paso hemos usado que $x^{\log_y z}$ es igual a $z^{\log_y x}$.

- En efecto, si aplicamos $\log_y$ a ambos, obtenemos
- $\log_y(x^{\log_y z})$ y $\log_y(z^{\log_y x})$, que luego de simplificar quedan
- $(\log_y x)(\log_y z)$ y $(\log_y z)(\log_y x)$ que son iguales.
- Como $\log_y$ es inyectiva, $x^{\log_y z} = z^{\log_y x}$ vale.

Volvamos a los cálculos.

Finalizando

$$t(n) \leq n^{\log_b a} c + dn^k (r^{m-1} + \ldots + r + 1)$$

donde $r = a/b^k$

- si $r = 1$, entonces $a = b^k$ y $\log_b a = k$ y además

$$\begin{aligned} t(n) &\leq n^k c + dn^k \log_b n \\ &\text{es del orden de } n^k \log n \text{ para } n \text{ potencia de } b \end{aligned}$$

- si $r \neq 1$, entonces

$$t(n) \leq n^{\log_b a} c + dn^k (\frac{r^m - 1}{r - 1})$$

Finalizando

caso $r > 1$

- si $r > 1$, como $r = a/b^k$ entonces $a > b^k$ y $\log_b a > k$ y además

$$
\begin{aligned}
t(n) &\leq n^{\log_b a}c + dn^k\left(\frac{r^m-1}{r-1}\right)\\
&\leq n^{\log_b a}c + \frac{d}{r-1}n^k r^m\\
&= n^{\log_b a}c + \frac{d}{r-1}n^k \frac{a^m}{(b^k)^m}\\
&= n^{\log_b a}c + \frac{d}{r-1}n^k \frac{a^m}{(b^m)^k}\\
&= n^{\log_b a}c + \frac{d}{r-1}n^k \frac{a^m}{n^k}\\
&= n^{\log_b a}c + \frac{d}{r-1}a^m\\
&= n^{\log_b a}c + \frac{d}{r-1}a^{\log_b n}\\
&= n^{\log_b a}c + \frac{d}{r-1}n^{\log_b a}\\
&\quad \text{es del orden de } n^{\log_b a} \text{ para } n \text{ potencia de } b
\end{aligned}
$$

Finalizando

caso $r < 1$

- si $r < 1$, como $r = a/b^k$ entonces $a < b^k$ y $\log_b a < k$. Además, $r - 1$ y $r^m - 1$ son negativos, para evitar confusión escribimos $\frac{1-r^m}{1-r}$ en vez de $\frac{r^m-1}{r-1}$.

$$
\begin{aligned}
t(n) &\leq n^{\log_b a} c + dn^k \left(\frac{1-r^m}{1-r}\right) \\
&= n^{\log_b a} c + \frac{d}{1-r} n^k (1 - r^m) \\
&\leq n^{\log_b a} c + \frac{d}{1-r} n^k \\
& \text{es del orden de } n^k \text{ para } n \text{ potencia de } b
\end{aligned}
$$

Conclusión

- Si

$$t(n) = \begin{cases} c & \text{si la entrada es pequeña o simple} \\ a * t(n/b) + g(n) & \text{en caso contrario} \end{cases}$$

- con $g(n)$ del orden de n^k,
- demostramos

$$t(n) \text{ es del orden de } \begin{cases} n^{\log_b a} & \text{si } a > b^k \\ n^k \log n & \text{si } a = b^k \\ n^k & \text{si } a < b^k \end{cases}$$

- para n potencia de b.

Extendiendo el resultado a todo n

- Hemos calculado el orden de cualquier algoritmo divide y vencerás, para n potencia de b.
- Queremos calcular el orden para todo n.
- Se puede comprobar que si
  - $t(n)$ es no decreciente, y
  - $t(n)$ es del orden de $h(n)$ para potencias de b para cualquiera de las tres funciones $h(n)$ que acabamos de considerar,

  entonces $t(n)$ es del orden de $h(n)$ (para n arbitrario, no solamente las potencias de b).
- es decir, el resultado puede extenderse para n arbitrario.

Recurrencias divide y vencerás

- Si

$$t(n) = \begin{cases} c & \text{si la entrada es pequeña o simple} \\ a * t(n/b) + g(n) & \text{en caso contrario} \end{cases}$$

  si $t(n)$ es no decreciente, y $g(n)$ es del orden de n^k, entonces

- $$t(n) \text{ es del orden de } \begin{cases} n^{\log_b a} & \text{si } a > b^k \\ n^k \log n & \text{si } a = b^k \\ n^k & \text{si } a < b^k \end{cases}$$

Ejemplo: búsqueda binaria

{Pre: $1 \leq$ lft $\leq$ n+1 $\wedge$ $0 \leq$ rgt $\leq$ n $\wedge$ a ordenado}
fun binary_search_rec (a: **array**[1..n] **of T**, x:**T**, lft, rgt : **nat**) **ret** i:**nat**
    var mid: **nat**
    if lft $>$ rgt $\rightarrow$ i = 0
      lft $\leq$ rgt $\rightarrow$ mid:= (lft+rgt) $\div$ 2
          if x $<$ a[mid] $\rightarrow$ i:= binary_search_rec(a, x, lft, mid-1)
            x $=$ a[mid] $\rightarrow$ i:= mid
            x $>$ a[mid] $\rightarrow$ i:= binary_search_rec(a, x, mid+1,rgt)
          fi
    fi
end fun
{Post: (i $=$ 0 $\Rightarrow$ x no está en a[lft,rgt]) $\wedge$ (i $\neq$ 0 $\Rightarrow$ x = a[i])}

Búsqueda binaria

Función principal

{Pre: n $\geq$ 0 }
fun binary_search (a: **array**[1..n] **of T**, x:T) **ret** i:**nat**
    i:= binary_search_rec(a, x, 1, n)
end fun
{Post: (i $=$ 0 $\Rightarrow$ x no está en a) $\wedge$ (i $\neq$ 0 $\Rightarrow$ x = a[i])}

Búsqueda binaria

Análisis

- Sea $t(n)$ = número de comparaciones que hace en el peor caso cuando el arreglo tiene n celdas.
- $$t(n) = \begin{cases} 0 & \text{si } n = 0 \\ t(n/2) + 1 & \text{si } n > 0 \end{cases}$$
- $a = 1$, $b = 2$ y $k = 0$.
- $a = b^k$.
- $t(n)$ es del orden de $n^k \log n$, es decir, del orden de $\log n$.

Búsqueda binaria usando DyV en python

Caso base y llamada principal

```
def simple(x):
    e,c,lft,rgt = x[0],x[1],x[2],x[3]
    mid = (lft+rgt) // 2
    return lft > rgt or e == c[mid]

def ad_hoc(x):
    lft,rgt = x[2],x[3]
    found = (lft+rgt) // 2
    if lft > rgt:
        found = -1
    return found
```

```
def bsearch(e,a):
    z = DyV((e,a,0,len(a)-1))
    return z

h = bsearch(5,[2,5,7,9])
```

Búsqueda binaria usando DyV en python

Caso recursivo

```
def decompose(x):
    e,c,lft,rgt= x[0],x[1],x[2],x[3]
    mid = (lft+rgt) // 2
    if e < c[mid]:
        xs = [(e,c,lft,mid-1)]
    else:
        xs = [(e,c,mid+1,rgt)]
    return xs

def combine(ys):
    return ys[0]
```

Análisis de algunos algoritmos

- Ordenación por selección es del orden de n^2.
- Ordenación por inserción es del orden de n^2 (peor caso y caso medio).
- Ordenación por intercalación es del orden de $n \log_2 n$.
- Ordenación rápida es del orden de $n \log_2 n$ (caso medio).
- Búsqueda lineal es del orden de n.
- Búsqueda binaria es del orden de $\log_2 n$.

¿Cómo comparar los órdenes de los algoritmos?

- Hay funciones que crecen más rápido que otras (cuando n tiende a $+\infty$).
- Escribiremos $f(n) \sqsubset g(n)$ para decir que $g(n)$ crece más rápido que $f(n)$. Por ejemplo:
  - $n \log_2 n \sqsubset n^2$.
  - $\log_2 n \sqsubset n$.
- Escribiremos $f(n) \approx g(n)$ para decir que $f(n)$ y $g(n)$ crecen al mismo ritmo. Por ejemplo:
  - $\frac{n^2}{2} - \frac{n}{2} \approx n^2$.
  - $45n^2 \approx n^2$.

Algunas condiciones

- No nos interesan las constantes multiplicativas:
  - $\frac{1}{4}n^2 \approx n^2$
  - $4n^3 \approx n^3$
  - $1000 \log n \approx \log n$
  - $\pi n \approx n$
- No nos interesan los términos menos significativos, que crecen más lento:
  - $n^2 + n \approx n^2$
  - $n^3 + n^2 \log_2 n \approx n^3$
  - $\log n + 3456 \approx \log n$
  - $n + \sqrt{n} \approx n$

¿Cómo comparar funciones según su crecimiento?

- Regla del límite. Sean $f(n)$ y $g(n)$ tales que
  - $\lim_{n \to +\infty} f(n) = \lim_{n \to +\infty} g(n) = \infty$, y
  - $\lim_{n \to +\infty} \frac{f(n)}{g(n)}$ existe.

  Entonces
  - si $\lim_{n \to +\infty} \frac{f(n)}{g(n)} = 0$, entonces $f(n) \sqsubset g(n)$.
  - si $\lim_{n \to +\infty} \frac{f(n)}{g(n)} = +\infty$, entonces $g(n) \sqsubset f(n)$.
  - caso contrario (el límite es un número real positivo), $f(n) \approx g(n)$.

Jerarquía

$$1 \sqsubset \log_2 n \approx \log_3 n \sqsubset n^{0.001} \sqsubset n^{1.5} \sqsubset n^2 \sqsubset$$
$$\sqsubset n^5 \sqsubset n^{100} \sqsubset 1.01^n \sqsubset 2^n \sqsubset 100^n \sqsubset$$
$$\sqsubset 10000^n \sqsubset n! \sqsubset n^n$$

Propiedades

- Constantes multiplicativas no afectan.
- Términos de crecimiento despreciable no afectan.
- Sean $a, b > 1$, $\log_a n \approx \log_b n$.
- Sea $f(n) > 0$ para "casi todo $n \in \mathbb{N}$". Entonces:
  - $g(n) \sqsubset h(n) \iff f(n)g(n) \sqsubset f(n)h(n)$.
  - $g(n) \approx h(n) \iff f(n)g(n) \approx f(n)h(n)$.
- Sea $\lim_{n\to\infty} h(n) = \infty$. Entonces:
  - $f(n) \sqsubset g(n) \implies f(h(n)) \sqsubset g(h(n))$.
  - $f(n) \approx g(n) \implies f(h(n)) \approx g(h(n))$.

Jerarquía

$$1 \sqsubset \log(\log(\log n)) \sqsubset \log(\log n) \sqsubset \log n \sqsubset n^{0.001} \sqsubset$$

$$\sqsubset n \sqsubset n \log n \sqsubset n^{1.001} \sqsubset n^{100} \sqsubset 1.01^{n} \sqsubset$$

$$\sqsubset n^{100} * 1.01^{n} \sqsubset 1.02^{n} \sqsubset 100^{n} \sqsubset 10000^{n} \sqsubset$$

$$\sqsubset (n-1)! \sqsubset n! \sqsubset (n+1)! \sqsubset n^{n}$$