>> Divisão e Conquista

-Observamos que um algoritmo pode invocar outro algoritmo dentro dele mesmo, o que pode afetar fortemente no cálculo de passos. Para calcular isso, utilizaremos a técnica de divisão e conquista, subdividida em três fases:

1. Dividir: Trabalhar sobre a instância do problema para gerar sub-instâncias menores.

2. Conquistar: Resolver o problema para as sub-instâncias construidas. Normalmente é um trabalho recursivo.

3. Combinar: Trabalhar sobre as soluções obtidas para as sub-instancias de forma a construir a solução procurada.

Problema: Contar o numero de identidades em um vetor numérico. Uma identidade e um índice i tal que V[i] = i.

```
contador ←0
para i ←1,2..|V| faça
|  se V[i] = i então
|  |  contador ← contador + 1
retorne contador
```

*O algoritmo esta certo, mas não acontece divisão e conquista.

Algoritmo: IdentidadesDC(V, p,q)
Entrada: Vetor numérico V, índices p e q
Saída: Número de identidades em V[p...q]

```
se p > q entao:                                   +1
|  retorne 0
se p = q entao:                                   +1
|  retorne V[p] = p
meio ← piso((p + q) / 2)                          +2
c1 ← IdentidadesDC(V, p, meio)   +Fp(piso(n / 2))
c2←IdentidadesDC(V, meio+1, q) + Fp(teto(n / 2))
retorne c1 + c2                                   +2
```

Total de passos
6 + Fp(piso(n / 2)) + Fp(teto(n / 2))

*Fp : Representação do valor dessa recursão que vamos calcular.
*Agora sim. Podemos calcular os passos atraves de divisão e conquista.

Complexidade de Algoritmos Recursivos

Calculamos F(n) = divisão + conquista + combinação --Valor total da iteração

Fp(n) = {6 + Fp(piso(n / 2) + Fp(teto(n / 2)) , n > 1
{2 , n = 0
{3 , n = 1

Fp(n) = 6 + Fp(piso(n / 2) + Fp(teto(n / 2)
= 6 + Fp(n / 2) + Fp(n / 2) --supondo que n / 2 é potencia de 2 pra facilitar o calculo, afinal, buscamos apenas um valor aproximado
= 6 + 2Fp(n / 2)
= $6 + 2(6 + 2Fp(n / 2^2))$ -- +1 recursão
= $6 + 2 * 6 + 2^2(6 + 2Fp(n / 2$^3)) -- +1 recursão
= $6 + 2 * 6 + 2^2 * 6 + 2^3(6 + 2Fp(n / 2$^4)) -- +1 recursão
= (Somatório (i = 0 a k-1) de 2^i * 6) + 2^k * Fp(n / 2^k)

* n / 2^k = 1 (vai chegar em 1) → n = 2^k → k = lg n

Fp(n) = (Somatório (i = 0 a lg n-1) de 2^i * 6) + 2^lgn * 3 --o 3 veio do numero de passos do ultimo caso.
= 6*(Somatorio(i=0 a lgn-1) de 2^i) + n*3 --lembrando que 2^lgn = 2^log2^2 = 1
= 6*2^lgn + 3n
= 6n + 3n
= Θ(n)

MS(V,p,q)

Se p < q entao: --+1
| meio ← piso((p + q) / 2) --+1
| MS(V, p, meio) --Fp(?)
| MS(V, meio+1, q) --Fp(?)
| Entrelaçamento(V, p, m, q) --Fp(n)<=c*n

Usando o método de arvore de recursão para recorrências nesse algoritmo MS:

nível	Fp(n)	
0	Fp(n/2) + Fp(n/2)	+ c*n
1	Fp(n/4)+Fp(n/4)+Fp(n/4)+Fp(n/4)	+ c*n/2 + c*n/2
2	8*Fp(n/8)	+ c*n
3	16*Fp(n/16)	+ c*n
4	.	
5	.	
6	.	
k	2^k*F(1) -.	-- nao tem c*n, ou seja, ele vai acontecer k-1 vezes

|
'----> n/2^k = 1 -> k = lg n

Fp(n) = (k – 1)c * n + 2^k * F(1)
Fp(n) = (lg n-1)c * n + n * 1
Fp(n) = c * n * lg n - c * n + n
= c * n * lg n -n * (1-c) = Θ(n * lg n)

Exemplo: $2 * T(n / 3) + n$

Usando o método de para recorrências:

Nível	#Termo	Recorrência	Não-recorrência
1	2^1	T(n/3)+T(n/3)	+n*(2/3)^0
2	2^2	2*T(n/9)+2*T(n/9)	+n*(2/3)^1
3	2^3	8*T(n/27)	+n*(2/3)^2
4	2^4	16*T(n/81)	+n*(2/3)^3
.			
.			
.			
k	2^k	2^k*T(1)	+n*(2/3)^k-1

T(n) = 2^k*c + Somatório(i = 0, k-1)n*(2/3)^i
T(n) = c*2^($\log_3^n$) + n * Somatório(i = 0, $\log_3^n$-1)n*(2/3)^i
<= c*2^($\log_3^n$) + n * Somatório(i = 0, ∞)n*(2/3)^i
<= c*2^($\log_3^n$) + n * 3
<= c*(3^$\log_2^3$)^($\log_3^n$) + n * 3
<= c*(3^$\log_3^n$)^($\log_2^3$) + n * 3
<= c*n^($\log_3^2$) + n * 3
= Θ(n)

Exemplo: T(n / 2) + T(n / 3) + n → T(n) = Ω(n)

Nível	Termo	Não-Recorrencia
1	T(n/2) + T(n/3)	n
2	T(n/4) + T(n/8) + T(n/8) + T(n/9)	(5/6)*n
3	T(n/8) + T(n/12) + T(n/12) + T(n/18) + T(n/12) + T(n/18) + T(n/18) + T(n/24)	(25/36)*n
.	.	.
.	.	.
.	.	.
k	2^k * T(1)	(5/6)^k-1 * n

T(n) <= T'(n) = 2^k * c + Somatório(i = 0, k-1)n*(5/6)^i
<= n * c + n * 6
= O(n)

Logo, como T(n) = Ω(n) e T(n) = O(n), então T(n) = Θ(n)

Suponhamos o seguinte vetor para QuickSort

| | | | | | |
'-------------> neste ultimo ficando o pivô, vimos que seria Θ(n^2)

Supondo um outro dessa forma
| | |99%| | | | 1%| -- Aqui, temos que 1% do vetor estara a direita, 99% a esquerda
'---------> Pivô

Temos então:

Fp(n) = { Fp((99/100)*n + Fp(n/100) + c*n
{ 1 , caso base (n=1)

Analisando com arvore de recorrência:

Nível	Recorrência	
0	Fp(n)	+ 0
1	Fp(99/100) + Fp(n/100)	+ c*n
2	$Fp((99/100)^2*n) + Fp((99/100^2)*n) + Fp(99/100^2*n) + Fp(n/100^2)$	+ 2c*n
.	. .	
.	. .	
.	. .	
k	2^k*Fp(1)	+ k*c*n

neste caso, como uma parte da recorrência terminara depois, e estamos buscando saber o valor de O, vamos calcular o O como se a arvore terminasse perfeita onde toda recorrência iria ate o nível da recorrência mais baixa.

tendo que: 1 = n/(100/99)^k -> n = (100/99)^k -> k = log(100/99)(n)   j = (100/99)

calculando: 2^k*Fp(1) + k*c*n = $2\hat{}(\log_j^n) + (\log_j^n * c * n$

$= ((100/99)\hat{}\log_j^2)\hat{}\log_j^n + \log_j^n * c * n$

$= n\hat{}\ \log_j^n + c * n * \log_j^n$

= O(n*logn)

TEOREMA MESTRE

Dada uma recorrência T(n) da forma:

T(n) = a * T(n / b) + Θ(n^c)   , n >= n0

k -> Caso base(Uma constante) , n < n0

Com a, b, c, n0, k constantes, b /= 0, temos:

(1) T(n) = Θ(n^$\log_b^a$)   , se $\log_b^a > c$

(2) T(n) = Θ(n^c)   , se $\log_b^a < c$

(3) T(n) = Θ(n^c * log n)   , se $\log_b^a = c$

Exemplo:

Bp(n) = { Θ(n^0) + 1*Bp(n / 2)
{ k

a = 1 , b = 2 , c = 0

$\log_2^1 = 0$

Bp(n) = Θ(lg n)

O que ter em mente para gerar um algoritmo de Divisão e Conquista?

1. É possível responder o problema em termo das respostas para sub-instâncias?
2. É "fácil" gerar sub-instâncias?
3. É "fácil" combinar as respostas das sub-instâncias na resposta buscada?

POTENCIAÇÃO COM EXPOENTE NATURAL

Calcular x^n, onde x pertence aos reais e n aos naturais.
(Supondo que x e n não são nulos simultaneamente)

x^n = { x^(n / 2) * x^(n / 2) , se n é par
{ x^((n – 1) / 2) * x^((n – 1) / 2) * x , se n é impar
{ x^1 = x
{ x^0 = 1

Algoritmo: PotNatural(x,n)
Entrada: x pertence aos reais e n aos naturais
Saída: valor de x^n
Requisito: x = 0 <=> n/=0

```
se n = 0 entao:
| retorne 1
se n é par entao:
| pot ←  PotNatural(x, n/2)
| retorne pot * pot
senao:
| pot ← PotNatural(x, (n – 1) / 2)
| retorne pot * pot * x
```

Pp(n) = { Pp(n / 2) + Θ(1) --> c = 0, b = 2, a = 1 → $\log_2 1$ = 0 → Pp(n) = Θ(n * lgn)
{ k

Fp(n) = { Fp((99/100)*n + Fp(n/100) + c*n
{ 1 , caso base (n=1)

Nível	Recorrência	Não-Recorrência
0	Fp(n)	+ 0
1	Fp(99/100) + Fp(n/100)	+ c*n
2	Fp((99/100)²*n) + Fp((99/100²)*n) + Fp(99/100²*n) + Fp(n/100²)	+ 2c*n
.	.	.
.	.	.
.	.	.

- Como esse termo ficou muito grande, e não conseguimos concluir perfeitamente o calculo, vamos tratar de um modo diferente.
- Vemos que, cada nível acima e exatamente a soma das das folhas do nível abaixo dele. Assim, vemos que o número de folhas é n.
- Não da pra saber a soma dos valores constantes.
- Mas, como buscamos um limite superior podemos tomar k*c*n onde k refere-se ao nível mais baixo.

.	.	.
k	n*Fp(1)	+ k*c*n

*Ja vimos anteriormente que

k	n*Fp(1)	+ log(100/99)*c*n

= log(100/99)(n)
= n*Fp(1) + log(100/99)*c*n
= c1*n+c*n*log(100/99)(n)
= O(n*logn)
