Começar pelos algs. recursivos p/ exponenciação !!

Projeto e Análise de Algoritmos (PAA)

Divisão e Conquista, MergeSort, Árvore de Recorrência

Etapas do projeto de algoritmos por divisão e conquista

deixar escrito no canto

- Dividir: o problema original é dividido em subproblemas menores do mesmo tipo.
- Conquistar: os subproblemas são resolvidos recursivamente, sendo que os subproblemas pequenos são caso base.
- Combinar: as soluções dos subproblemas são combinadas numa solução do problema original.

Exemplo/ideia do mergeSort:

- vetor fora de ordem v 5 4 1 3 | 6 2 7
- dividir o vetor a ser ordenado em dois subvetores, cada um com metade do tamanho original vl 5 4 1 3 vr 6 2 7
- ordenar recursivamente os dois subvetores, sendo que subvetores com 0 ou 1 elementos já estão ordenados vl 1 3 4 5 vr 2 6 7
- intercalar (merge) os subvetores ordenados resultantes

v 1 2 3 4 5 6 7

Pseudocódigo do algoritmo mergeSort recursivo:

não apagar

```
mergeSort(v):
    se n ≤ 1 : retorna
    (v_e, v_r) = metades esquerda e direita de v
    mergeSort(v_e); mergeSort(v_r)
    v = intercala(v_e, v_r)
```

Pseudocódigo do algoritmo intercala:

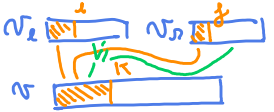
```
intercala(v_e, v_r):                    n_e = tam(v_e); n_r = tam(v_r); n = tam(v)
    i = j = k = 1
    enquanto (i ≤ n_e E j ≤ n_r):                                   O(n_e + n_r)
        se v_e[i] ≤ v_r[j] : v[k] = v_e[i]; k++; i++;
        senão /* v_r[j] < v_e[i] */ v[k] = v_r[j]; j++; k++;
    laços p/ copiar o que sobrou em v_e ou v_r
    devolva v
```

Eficiência do intercala: o número de operações é linear no tamanho dos subvetores,

- i.e., da ordem de $\Theta(n_e + n_r) = O(n)$
- Para verificar, note que em cada iteração um elemento é colocado em v.

(*) adiaram 3 depois sob demanda

Corretude do intercala com invariantes de laço: no início de cada iteração



1 • $v[1..k-1]$ contém os elementos de $v_e[1..i-1] \cup v_r[1..j-1]$

2 • os elementos de $v[1..k-1]$ estão ordenados,

3 • $v[1..k-1] \leq v_e[i..n_e]$ e $v[1..k-1] \leq v_r[j..n_r]$ (1)

Base: $i = j = k = 1 \Rightarrow [v[1..k-1] = v[1..0] = \phi] = v_e[1..i-1] \cup v_r[1..j-1]$ (2)
e 3 vale pois $v[1..k-1] = \phi$ não existindo elemtos p/ violar as desigualdades

H.I.: no início da iteração atual do alg. temos [$v[1..k-1]$ está ordenado] 2
e 1 [$v[1..k-1] = v_e[1..i-1] \cup v_r[1..j-1]$] e [$v[1..k-1] \leq v_e[i..n_e]$ e $\leq v_r[j..n_r]$] 3

Destaque: Passo: sejam i', j' e k', respectivamente, as variáveis i, j e k ao final da iteração atual do alg. Queremos mostrar que os invariantes 1, 2 e 3 valem p/ i', j' e k'.

Durante a iter. atual temos dois casos a) $v_e[i] \leq v_r[j]$ e b) $v_e[i] > v_r[j]$

(caso b é simétrico) – Se a) então $v[k] = v[i]$, $i' = i+1$, $j' = j$ e $k' = k+1$. Assim

restaurou 1 [$v[1..k'-1] = v[1..k] \overset{\text{1 da H.I. + caso a)}}{=} v_e[1..i] \cup v_r[1..j-1] = v_e[1..i'-1] \cup v_r[1..j'-1]$]

restaurou 2 [e $v[1..k'-1] = v[1..k]$ está ordenado ??? Sim, graças a 2 e 3 na H.I.]

restaurou 3 [e $v[1..\overset{k}{k'-1}] \leq v_e[\overset{i+1}{i'}..n_e]$ e $v[1..\overset{k}{k'-1}] \leq v_r[\overset{j}{j'}..n_r]$ graças a 3 na H.I., ao fato de v_e estar ordenado, e ao caso a)]

Quiz: prova caso b). // Ao final os invariantes garantem que $v[1..k-1]$ está ordenado e só ñ contém os elemtos do sufixo de v_e ou v_r, que são $\geq$ aos elemtos em v

Pseudocódigo do algoritmo mergeSort recursivo:

```
mergeSort(vetor v):
    se tam(v) <= 1: devolva
    (vl, vr) = 1ª e 2ª metade de v, respectivamente
    mergeSort(vl)
    mergeSort(vr)
    v = intercala(vl, vr)
```

Corretude do mergeSort por indução:

- Pelo caso base, sabemos que nosso algoritmo devolve subvetores ordenados quando estes têm tamanho menor ou igual a 1.
- Supondo, como H.I., que nosso algoritmo ordena corretamente
  - um subvetor de tamanho n/2,
- verificamos que ele ordena um vetor de tamanho n,
  - uma vez que a função intercala funciona corretamente.

Análise de eficiência:

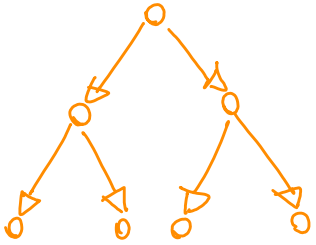
- Quantos são os subproblemas/chamadas recursivas? 2
- Qual o tamanho dos ~~inteiros~~ vetores nos subproblemas? n/2
- Qual o trabalho local (não recursivo)? $c.n = O(n)$
- Assim, o tempo gasto é dado pela recorrência: $T(n) = 2T(n/2) + c.n$
- Mas, qual é a função de tempo deste algoritmo?

Quiz: resolva a recorrência $T(n) = 2T(n/2) + c.n$ pelo método da substituição

Resolvendo a recorrência T(n) = 2T(n/2) + cn

- usando uma árvore (binária) de recursão.

nível	# problemas	tam. subproblema
0	1	n
1	2	$n/2$
2	4	$n/4$
⋮	⋮	⋮
i	2^i	$n/2^i$



Qual o último nível h da árvore?

- Note que, as chamadas recursivas param quando o tamanho do subproblema é 1. Portanto $n/2^h = 1 \Rightarrow 2^h = n \Rightarrow h = \lg n$

Como começamos a contar os níveis no nível 0, o número de níveis é $1 + \lg n$

O número de níveis da árvore é $1 + \lg n$ já que:

- no nível 0 temos n elementos no vetor,
- lg n = log_2 n é o número de vezes que podemos dividir n por 2 antes dele se tornar menor ou igual a 1 (caso base).

No nível j o número de subproblemas é 2^j

- e o tamanho do vetor dos subproblemas é $n/2^j$

Qual o trabalho local (não recursivo) do mergeSort?

- Uma chamada do mergeSort realiza basicamente um teste,
  - seguido de duas chamadas recursivas e uma chamada de intercala.
- Como intercala tem eficiência de tempo linear no tamanho dos vetores,
  - o trabalho não recursivo num vetor de tamanho m é $c.m$

Qual o trabalho realizado no nível j?

$$\text{Trab\_nível}(j) = \#\text{subprob.} * \text{trab\_por\_subprob}$$

$$= 2^j . c . n/2^j = c.n$$

Por fim, o trabalho total é dado pela soma do trabalho por nível da árvore

- ao longo do número de níveis da árvore, i.e.,

$$Trab\_total = \sum_{i=0}^{\lg n} Trab\_nível(i) = \sum_{i=0}^{\lg n} c.n = c.n(1+\lg n) = \Theta(n \lg n)$$

Numa comparação rápida, para n = 10^6 e 10^9 temos:

- lg n = log_2 n
- n lg n = n log_2 n
- n^2

	20	30
	2.10^7	3.10^{10}
	10^{12}	10^{18}

Pulan

Supondo que um computador realize 1 Giga (10^9) operações por segundo, temos:

- um algoritmo de ordenação O(n log n) leva, da ordem de, centésimos de segundo e 30 segundos,
- um algoritmo de ordenação O(n^2) leva, da ordem de, 16 minutos e 32 anos.

Animação: https://www.youtube.com/watch?v=ZZuD6iUe3Pc

Extra: projetar e analisar função exp(b,n) que devolve b^n

Versão 1: $exp(b,n) = b * exp(b,n-1) \Rightarrow T(n) = T(n-1) + c = \Theta(n)$ $\Rightarrow (\lg b^n)^2 = n^2 \lg^2 b \Rightarrow \Theta(n^3)$

Versão 2: $exp(b,n) = exp(b,n/2)^2 \Rightarrow T(n) = T(n/2) + c = \Theta(\lg n)$ $\Rightarrow (\lg b^n)^2 = n^2 \lg b \Rightarrow \Theta(n^2)$

(Bônus) Resolvendo a recorrência por substituição: T(n) = 2 * T(n/2) + cn

$T(n) = 2T(n/2) + cn$

$T(n/2) = 2T(n/4) + cn/2$

$T(n/4) = 2T(n/8) + cn/4$

$T(n/8) = 2T(n/16) + cn/8$

$T(n) = 2T(n/2) + cn = 2^1 T(n/2^1) + 1cn$

$= 2(2T(n/4) + cn/2) + cn$

$= 4T(n/4) + 2cn = 2^2 T(n/2^2) + 2cn$

$= 4(2T(n/8) + cn/4) + 2cn$

$= 8T(n/8) + 3cn = 2^3 T(n/2^3) + 3cn$

$= 8(2T(n/16) + cn/8) + 3cn$

$= 16T(n/16) + 4cn = 2^4 T(n/2^4) + 4cn$

$T(n) = 2^j T(n/2^j) + jcn$

no maior j temos $\frac{n}{2^j} = 1 \Rightarrow \lg(2^j) = \lg n \Rightarrow j = \lg n$

$T(n) = 2^{\lg n} \cdot T(1) + \lg n \cdot c \cdot n = n \cdot c + cn \lg n$

$$T(n) = 2^{\lg n} \cdot T(1) + \lg n \cdot c \cdot n = n \cdot c + c n \lg n$$

Vamos verificar que a fórmula que deduzimos está correta usando indução matemática

Caso base: $T(1) = 1 \cdot c + c \cdot 1 \cdot \lg 1 = c$ ✓

H.I.: $T(n') = cn' \lg n' + cn'$ p/ $n' < n$

Passo: $T(n) = 2T(n/2) + cn$

pela H.I. $= 2\left(c(n/2)\lg(n/2) + c\cdot(n/2)\right) + cn$

$= cn \lg n - cn \lg 2 + cn + cn$

$= cn \lg n + cn$ ✓

Portanto $T(n) = cn \lg n + cn = \Theta(n \lg n)$

Dados inteiros a e b, c/ $a \neq 0$ e $b \geq 0$, devolva a^b

↳ $b \in \mathbb{N}$

⊗ Apresentar no final da aula 2 sobre indução, ou no começo da aula 5 sobre D&C e mergeSort. Talvez faça o simples na 2 e o esperto na 5 !!!

1ª Tentativa:

$a^b = a^{b-1} \cdot a$ ⇒

resolva isso recursivamente (c/ cara nossa H.I.)

```
alg RecSimples (int a, nat b):
    se b = 0 devolva 1
    aux = alg RecSimples (a, b-1)
    devolva a . aux
```

Provando corretude usando ind. matemática

- Base: P/ $b = 0$ temos algRecSimples(a,b) devolve $1 = a^0 = a^b$ ✓
- H.I.: P/ $b' = b - 1$ temos algRecSimples(a,b') devolve $a^{b'} = a^{b-1}$
- Passo: P/ b temos algRecSimply(a,b) devolve $a \cdot aux \overset{H.I.}{=} a \cdot a^{b-1} = a^b$ ✓

Recorrência de tempo $T(a,b) = T(a,b-1) + c$ e $T(a,0) = c$

Expandindo

$T(b) = T(b-1) + c$

$T(b-1) = T(b-2) + c$

$T(b-2) = T(b-3) + c$

⋮

Substituindo

$T(b) = T(b-1) + c$

$= T(b-2) + 2c$

$= T(b-3) + 3c$

⋮

Generalizando

$T(b) = T(b-i) + ic$

Base da recursão

$T(b-i) = T(0) = c$

$b - i = 0 \Leftrightarrow b = i$

Finalizando $T(b) = T(b-i) + ic \overset{i=b}{=} T(0) + b.c = (b+1).c = O(b)$ = linear em b

↓

Note que bastam k bits para representar $b \leq 2^k - 1$

Por isso, este algoritmo $O(b) = O(2^k)$ é exponencial no tamanho de uma parte da entrada

Dados inteiros a e b, c/ $a \neq 0$ e $b \geq 0$, devolva a^b

↳ $b \in \mathbb{N}$

2ª tentativa:

$a^b = (a^{\lfloor b/2 \rfloor})(a^{\lfloor b/2 \rfloor})$ se b for par

$a^b = (a^{\lfloor b/2 \rfloor})(a^{\lfloor b/2 \rfloor}) \cdot a$ se b for ímpar

Exemplos:

- $b = 6 \quad a^6 = a^3 \cdot a^3$
- $b = 7 \quad a^7 = a^3 \cdot a^3 \cdot a$

```
alg Rec D&C (a, b):
    se b == 0 : devolva 1
    aux = alg Rec D&C(a, ⌊b/2⌋)
    aux = aux * aux
    se b % 2 == 1 : aux = aux * a
    devolva aux
```

Recorrência de tempo

$T(b) = 1T(b/2) + c \quad$ e $\quad T(0) = c$

Expandindo / Substituindo / Generalizando

$T(b) = T(b/2^i) + ic$

como a divisão de b por 2 é inteira, i.e., resultado é arredondado p/ baixo, na penúltima chamada recursiva $(h-1)$ temos:

$b/2^{h-1} = 1 \Rightarrow 2^{h-1} = b \Rightarrow h - 1 = \lg b \Rightarrow h = 1 + \lg b$

Portanto

$$T(b) = T\left(\frac{b}{2^{1+\lg b}}\right) + (1+\lg b)c = T\left(\left\lfloor \frac{1}{2} \right\rfloor\right) + (1+\lg b)c$$

$$= c + c + c\lg b = O(\lg b)$$

↳ como $\lg b = \lg 2^k = k$ este alg. é linear no # de bits k usado na entrada p/ representar b

Intuição