Desenho e Análise de Algoritmos

Notas - Aula TP 7 e 8

Análise de algoritmos recursivos
Continuação

Ana Maria de Almeida

DCTI
2022/2023

1

Plano para esta aula

- Fórmulas recorrentes para análise de complexidade
- Algoritmos clássicos do tipo “divide-and-conquer” e aplicações.

2022/2023

3

Nas últimas aulas vimos:

- Análise de algoritmos: as 3 questões essenciais
- Analisámos alguns algoritmos de ordenação:
  - Algoritmos quadráticos: Selection sort e Insertion sort
  - Algoritmo da fusão ordenada: *merge sort O(n log n)*
  - *Algoritmo quicksort, que é quadrático no pior dos casos*
- Estabelecemos mais algumas estratégias de desenho de algoritmos:
  - Pesquisa Exaustiva
  - Estratégia Incremental ou *diminuir* para conquistar (*decrease-and-conquer*) tendo como exemplos o Insertionsort e o Selectionsort
  - Vimos que o Mergesort e o Quicksort são ambos do tipo dividir para conquistar (*divide-and-conquer*)

4

recursão, indução e teoremas

continuação

6

Recorrência e Indução

- Terminologia: Recursão = Recorrência
- Em ambos os conceitos temos dois elementos construtores: condição (condições) de paragem ou caso(s) base e condição (condições) geral (gerais).
- Uma **condição geral divide** o problema em problema(s))**idêntico**(s) mas de **menor dimensão**
- Uma **condição de paragem**, ou caso base, não é recorrente e **pára** a recorrência de uma condição geral.
- A Indução Matemática permite, pelo mesmo racional, resolver (fechar) fórmulas recorrentes.

  Parte de um caso simples e é desenvolvido para casos mais complexos (bottom up?)

  Exemplo: Consideremos a fórmula recorrente

$$T_n = \begin{cases} 0 & \text{se } n = 0 \\ 2T_{n-1} + 1 & \text{se } n > 0 \end{cases}$$

Recorrência e Indução: fecho de uma fórmula recorrente

$$T_n = \begin{cases} 0 & \text{se } n = 0 \\ 2T_{n-1} + 1 & \text{se } n > 0 \end{cases}$$

n	0	1	2	3	4	5	6	7
T_n	0	1	3	7	15	31	63	127

Ora:

$T_n = 2T_{n-1} + 1$

= 2 (2Tn-1+1) + 1

= 2^2 Tn-1 + 1 + 2

= 2^3 Tn-3 + 1 + 2 + 2^2

$= \cdots =$

$= \cdots =$

$=$

$=$

Passo geral (k) da recorrência:
2^k Tn-k + 1 + 2 + 2^2 + 2^3 + ... + 2^(k-1)

A recorrência pára quando ...

n - k = 0 → n = k

2^n T0 + 1 + 2 + 2^2 + 2^3 + ... + 2^(n-1) =
= 2^n * 0 + Série Geométrica de razão a = 2
= 2^n - 1

Análise (mais) formal da complexidade do Mergesort

- Relembre que o algoritmo de Fusão Ordenada (*Merge*) é de ordem linear, *O(n)*.
- Estratégia "dividir-para-conquistar":
  1. Descendente (Top-down): Separar recursivamente em 2 sub-sequências de tamanho aproximadamente igual até obter os elementos individuais;
  2. Ascendente (Bottom-up): Fundir em ordem duas subsequências ordenadas


```
mergesort(a, ini, fim):
  if ini < fim:
    meio = maior inteiro contido em (ini+fim/2)
    mergesort(a, ini, meio)
    mergesort(a, meio+1,  fim)
    merge(a, ini, meio, fim)
  endif
```

10

Análise (mais) formal da complexidade do Mergesort

- O algoritmo de Fusão Ordenada é de ordem linear, *O(n)*.
- Em termos de trabalho total na parte da fusão, cada chamada recursiva vai efetuar $\frac{n}{2}$ comparações (para fundir em ordem) + n atribuições (colocação dos elementos na posição final):

```
mergesort(a, ini, fim):
  if ini < fim:
    meio = maior inteiro contido em (ini+fim/2)
    mergesort(a, ini, meio)
    mergesort(a, meio+1,  fim)
    merge(a, ini, meio, fim)
  endif
```

$$T(n) = \begin{cases} 1 & se\ n = 1 \\ 2T\left(\frac{n}{2}\right) + n & senão \end{cases}$$

- Logo, o "trabalho" do merge sort, em termos assintóticos é então:

Exercício de aula: usando o mesmo raciocínio do exemplo nos slides anteriores, prove que o trabalho do mergesort é $T(n) = O(n \log n)$.

11

2 T(n/2) + n = 2 (2T(n/4) + n/2) + n = 4 (2 T(n/8) + n/4) + n = 8 (2 T(n/8) + n/8) + n =
2^(k) T(n/2^k) + 2^(k)n + kn = nT(1) + n log2n = n + n log2n

Divide-and-conquer: fórmula geral de recorrência

- Um algoritmo do tipo *divide-and-conquer* resolve um problema dividindo uma instância de tamanho n em várias instâncias mais pequenas
- Assumindo que todas as menores instâncias são do mesmo tamanho, i.e, que diminuem num factor constante, seja n/b, e que são resolvidas 'a' instâncias menores, obtemos a seguinte fórmula recorrente:

Fórmula recorrente geral para a estratégia divide-and-conquer

$$T(n) = a\,T\left(\frac{n}{b}\right) + f(n)$$

Teorema Principal (para fórmulas recorrentes)

- Consideremos as constantes positivas: $a \geq 1, b \geq 2$
- Seja $\mathbf{f(n)} \leq \mathbf{O(n^d)}, d \geq 0$, a função que descreve o trabalho requerido para compor soluções de subproblemas.
- Assumimos ainda que, s.p.d.g., n é uma potência de b ($b^k, k = 1, 2, \ldots$).
- Então, a solução da recorrência $T(n) = \begin{cases} \Theta(1) & se\ n = 1 \\ a\,T\left(\frac{n}{b}\right) + f(n) & caso\ contrário \end{cases}$ é dada por:

a) Se $\boldsymbol{a} < \boldsymbol{b^d} \Longrightarrow T(n) = O(n^d)$
b) Se $\boldsymbol{a} = \boldsymbol{b^d} \Longrightarrow T(n) = O(n^d \log n)$
c) Se $\boldsymbol{a} > \boldsymbol{b^d} \Longrightarrow T(n) = O(n^{\log_b a})$

Aplicação Teorema Principal - Exemplos

$$T(n) = \begin{cases} \Theta(1) & se\ n = 1 \\ a\,T\left(\frac{n}{b}\right) + f(n) & caso\ contrário \end{cases}$$

a) Se $\boldsymbol{a} < \boldsymbol{b^d} \Rightarrow T(n) = O(n^d)$

b) Se $\boldsymbol{a} = \boldsymbol{b^d} \Rightarrow T(n) = O(n^d \log n)$

c) Se $\boldsymbol{a} > \boldsymbol{b^d} \Rightarrow T(n) = O(n^{\log_b a})$

- Em que casos é possível aplicar este teorema?

a) $T(n) = 9T(\frac{n}{3}) + 1$ — a = 9 ; b = 3 ; d = 0
a > b^d

b) $T(n) = 2T(\frac{n}{2}) + n$ — a = 2; b = 2; d = 1
a = b

c) $T(n) = 4T(\frac{n}{2}) + n^2$ — a = 4; b = 2; d = 2
a = b^d

d) $T(n) = 3T\left(\frac{n}{2}\right) + k\,n$

e) $T(n) = 4T\left(\frac{n}{2}\right) + n^3$

Notas sobre o Teorema Principal para fórmulas recorrentes: compreender o Teorema

Sobre o que representam os 3 parâmetros no Teorema Principal:

a : representa quantidade de subproblemas resultantes da divisão.

b : indica o factor de diminuição do tamanho do input).

d : quantidade de trabalho necessário para criar os subproblemas e/ou combinar soluções na construção da solução final.

- $a \geq 1, b > 1\ e\ d$ são constantes.
- Seja $T(n) = a \cdot T\left(\frac{n}{b}\right) + O(n^d)$. Então: $T(n) = \begin{cases} O(n^d \log(n)) & \text{if } a = b^d \\ O(n^d) & \text{if } a < b^d \\ O(n^{\log_b(a)}) & \text{if } a > b^d \end{cases}$

Fórmulas recorrentes - **Notas finais**

- Vamos caracterizar sempre um algoritmo recorrente através da classificação do trabalho/tempo desse algoritmo em termos de notação assintótica big-O.

- Para isso, vamos ter de formular uma descrição desse trabalho em todos os casos: geral (recorrente) e base (de paragem da recursão). Ou seja, formular uma fórmula do tipo

$$T(n) = \begin{cases} g(n) & se\ n\ é\ caso\ base \\ a\,T\left(\frac{n}{b}\right) + f(n) & caso\ contrário \end{cases}$$

onde:

- *f(n) é* uma função que descreve o trabalho requerido para compor soluções vindas de subproblemas numa solução para a dimensão do problema nessa chamada e
- g(n) é a função que descreve o número de operações no caso base (sem recursão).

Análise de algoritmos recorrentes

- Muitos dos algoritmos que existem (em particular do tipo divide-and-conquer) podem ser analisados/classificados, usando relações de recorrência para obter a sua ordem $O(.)$
- Exemplos: muitas das funções $a: \mathbb{N} \rightarrow \mathbb{R}$ são facilmente definidas como relações recorrentes:
  - $a_n = a_{n-1} + 1$ e $a_1 = 1$ equivale ao polinómio $P(n)=n$
  - $a_n = 2\,a_{n-1}$ e $a_1 = 1$ equivale à exponencial $f(n) = 2^{n-1}$
  - $a_n = n\,a_{n-1}$ e $a_1 = 1$ equivale ao factorial $f(n) = n!$
- É usual encontrar uma fórmula recorrente como solução de um problema de contagem, sendo necessário resolver (fechar) a recorrência, o que pode ser feito para estes casos se tomarmos em linha de conta que ...

a recorrência não é mais do que Indução Matemática!

Análise (mais) formal da complexidade do Quicksort

- Relembre que o processo de separação (ou de partição) é O(n).
- Como vimos, o peso computacional do Quicksort depende da separação efectuada em cada nível (do tamanho relativo de cada um dos sub–vectores criados para as chamadas recorrentes)

```
quicksort(a, ini, fim):

   if ini < fim:
      partir = partition(ini, fim)
      quicksort(a, ini, partir)
      quicksort(a, partir+1,  fim)
   endif
```

- Caso se consiga uma partição equilibrada, i.e, dividindo a sequência inicial em (aprox.) metade dos elementos para cada lado, cada chamada recursiva seria sobre n/2 elementos e, neste caso, o trabalho total será dado pela fórmula recorrente:

$$T(n) = \begin{cases} 1 & se\ n = 1 \\ 2T\left(\frac{n}{2}\right) + k\,n & se\ n > 1 \end{cases}$$

- **Neste caso,** aplicando o teorema principal temos: $T(n) = O(n \log n)$.

Como encontrar a Mediana

- Garantimos que o Quicksort vai separar sempre a sequência em duas metades se escolhermos como pivot de separação a **mediana** dos valores da sequência.
- Como escolher a mediana de uma sequência não ordenada?
  - Escolher o (n/2)-ésimo menor elemento da sequência.
- Ou seja, devemos resolver o problema da **K-Seleção**:
  - **Input**: uma sequência de tamanho n e um valor inteiro positivo 1 ≤ k ≤ n.
  - **Output**: o k-ésimo menor elemento da sequência
- e, para k = n/2 temos a mediana da sequência

7	4	3	8	1	5	9	14

- kSELECT(A, 1) = min(A)
- kSELECT(A, n/2) = mediana(A)
- kSELECT(A, n) = max(A)

Algoritmo 1:

- Se ordenarmos a sequência, o k-ésimo valor menor estará na posição de índice k.

```
Select(A, k):
    //Input: sequência, A, e inteiro k
   //Output: o k-ésimo menor valor de A

    A = Mergesort(A)
    return A[k]
```

- O tempo/trabalho computacional deste algoritmo será O(n log n).
- Podemos melhorar este **benchmark**?

Resolver o problema da **k-Seleção**

- Podemos tentar uma abordagem semelhante à da *Separação de Hoare*:
  - escolhido um pivot, separar os menores para a esquerda e os maiores para a direita
  - Exemplo:

pivot: 5

2	36	5	21	8	13	11	20	5	4	1

2	4	1

5	5

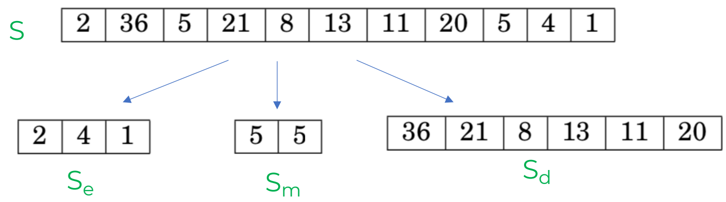
36	21	8	13	11	20

"Onde" está situado o mínimo/máximo elemento da sequência?

"Onde" está situado o sétimo menor elemento da sequência?

- O k-ésimo menor elemento tem de se situar após o índice k-1 após a separação.
- Esta abordagem permite escolher apenas uma das partes para procurar o elemento requerido, estudando apenas os tamanhos da lista

k-Seleção



- Ou seja:

$$select(S,k) = \begin{cases} seleção(S_e,k) & se\ k \le |S_e| \\ pivot & se\ |S_e| < k \le |S_e| + |S_m| \\ seleção(S_d,k) & senão \end{cases}$$

$|S_x|$ = tamanho da sequência S_x

descrição da divisão recursiva do problema inicial em subproblemas

- O que está errado na descrição de select(S, k) acima?

k-seleção: algoritmo final

k-Seleção

```
k-Select(A, k):
    //Input: sequência, A, e inteiro k
    //Output: o k-ésimo menor elemento de A

    if len(A) < 50:
       A = Mergesort(A)
       return A[k]
    p = getpivot(A)
    L, m, R = Partitition(A, p)
    if len(L)== k-1: return A[m]
    else if len(L) > k-1: return k-Select(L, k)
    else if len(L) < k-1: return k-Select(R, k-Len(L)-1)
```

As 3 questões essenciais

- Correção: "Está correcto?"
  - Com testes empirícos parece funcionar.
  - No pior dos casos, fazemos todas as separações possíveis até encontrar um pivot que é o elemento procurado.
- Análise de eficiência: "É rápido?"

  Fazer a análise assintótica suficientemente geral. i.e., análise do pior dos casos.
- Otimização: Podemos melhorar?

Análise de complexidade

Assumindo que o pivot é escolhido em tempo O(n)...

- $$T(n) = \begin{cases} T(\boldsymbol{tamanho\ de\ L}) + O(n) & len(L) > k - 1 \\ T(\boldsymbol{tamanho\ de\ R}) + O(n) & len(L) < k - 1 \\ O(n) & len(L) = k - 1 \end{cases}$$

- Como vimos na análise do quicksort, ***tamanho de L*** (len(L)) e ***tamanho de R*** (len(R)) dependem da escolha do *pivot.*
- O melhor desempenho (mais rápido, com menor peso computacional) seria escolher um pivot que **garanta** que len(L) = k-1.
- No entanto, não controlamos o valor k e apenas podemos decidir como se escolhe o pivot.

k-Seleção: escolha de pivot e impacto computacional

- Se conseguirmos particionar a sequência em metade para a esquerda e metade para a direita, a ordem de complexidade geral pode ser calculada usando a fórmula recursiva $T(n) = T\left(\frac{n}{2}\right) + O(n)$.
- Pelo teorema principal (porque a = 1 < b = 2 e d = 1) **poderíamos** concluir que este seria um algoritmo de ordem linear .

por enquanto, estamos apenas a assumir/colocar hipóteses e a investigar, nada mais!

- Para garantir este tipo de partição, seria necessário usar como pivot …

Escolha de um pivot "suficientemente" bom

- De facto, um pivot que divida aproximadamente em

3n/10 < len(L) < 7n/10 e 3n/10 < len(R) < 7n/10

teria um trabalho computacional, no pior dos casos:

$$T(n) \le T\left(\frac{7n}{10}\right) + O(n)$$

aplicando o Teorema principal, como a = … < b = …, d = …, temos :

$$T(n) \le O(\dots) = O(\dots)$$

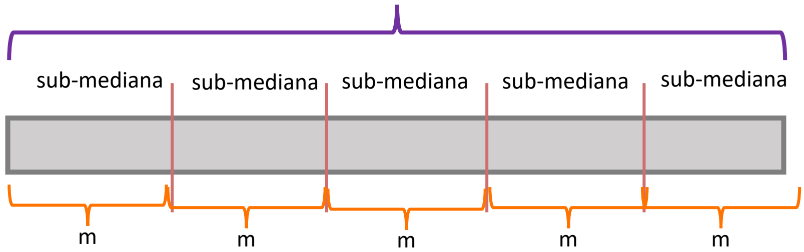
Seja $T(n) = a \cdot T\left(\frac{n}{b}\right) + O\left(n^d\right)$. Então

$$T(n) = \begin{cases} O(n^d \log(n)) & \text{if } a = b^d \\ O(n^d) & \text{if } a < b^d \\ O(n^{\log_b(a)}) & \text{if } a > b^d \end{cases}$$

Solução divide-and-conquer

- Ainda não conseguimos resolver a escolha da mediana de A (k = n/2).
- Mas podemos *dividir e conquistar*, resolvendo k-Select(X, m/2) para **valores suficientemente pequenos** de m (len(X) = m.)
- **Lema:** A mediana das sub-medianas aproxima a mediana da totalidade dos valores.

que vamos usar como pivot → mediana das sub-medianas ≈ mediana do total ← *que seria o pivot ideal*



Algoritmo divide-and-conquer para escolher um pivot "suficientemente bom"

- **choosePIVOT**(A):

```
Dividir A em m = ⌈n/5⌉ grupos de tamanho <= 5 cada.
for i=1, .., m:
    Calcular a mediana do grupo i, pi
p = SELECT( [ p1, p2, p3, …, pm ] , m/2 )
devolver o índice de p em A
```

Autores: Blum, Floyd, Pratt, Rivest, Tarjan (1973)

tempo O(1) uma vez que cada grupo tem tamanho 5. Logo, O(m)=O(n) para o ciclo completo.

- Este algoritmo consegue aproximar a divisão em duas metades da sequência. De facto, prova-se que:

Lema 1: usando o algoritmo anterior tem-se

$$|L| \leq \frac{7n}{10} + 5 \quad \text{e} \quad |R| \leq \frac{7n}{10} + 5$$

Escolha aleatória de pivot

- **Escolher aleatoriamente o pivot entre os elementos da lista**.
- O pior dos casos será quando a escolha aleatória recai **sempre** no maior ou no menor elemento, caso em que a ordem de complexidade deste algoritmo seria $\Theta(n^2)$.
- O melhor caso seria (ter a sorte de) escolher um pivot que separasse **sempre** em metade e teríamos O(n).
- Vamos definir um pivot como sendo **"bom"** se estiver entre os percentis 25 e 75 dos valores (entre o 1.º e o 3.º quartis): assegurando que as subsequências. S_e e S_d terão tamanho, no máximo, 3/4 do tamanho de S.

Escolha **aleatória** de pivot

- Em probabilidade, metade (50%) dos valores de S estarão entre os percentis 25 e 75.
- Com estas probabilidades, quantas tentativas temos de fazer para assegurar um valor de pivot "bom"?

k-seleção: algoritmo final

k-Seleção

```
k-Select(A, k):
    //Input: sequência, A, e inteiro k
    //Output: o k-ésimo menor elemento de A
    if len(A) < 10:
        A = Mergesort(A)
        return A[k]
    p = choosePivot(A)
    L, m, R = Partition(A, p)
    if len(L)==k-1: return A[m]
    else if len(L) > k-1: return k-Select(A, k)
    else if len(L) < k-1: return k-Select(R, k-Len(L)-1)
```

Qual a ordem de complexidade do algoritmo especial de seleção?

Quão “rápido” é?

```
Select(A, k):
    //Input: sequência, A, e inteiro k
    //Output: o k-ésimo menor elemento de A
    p = choosePivot(A)
    L, m, R = Partitition(A, p)
    if len(L)==k-1: return A[m]
    else if len(L) > k-1: return Select(A, k)
    else if len(L) < k-1: return Select(R, k-Len(L)-1)
```

- Usando o Lema 1 referido num slide anterior temos que

$$|L| \leq \frac{7n}{10} + 5 \text{ e } |R| \leq \frac{7n}{10} + 5$$

- A relação de recorrência para calcular o trabalho computacional do algoritmo da seleção (SELECT), **usando esta escolha de pivot**, será, então

$$T(n) \leq T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right) + O(n)$$

- O teorema principal não pode ser aplicado nesta recorrência (porquê?). Deve ser usado o método da substituição ou fecho da recorrência.

O método da substituição

- Passo 1: Conjecturar (adivinhar!) qual a possível conclusão.
- Passo 2: Provar, usando indução, que a conjectura é válida.

Hipótese Indutiva para n: existe algum $n_0 > 0$ e algum $C > 0$ tal que, $\forall n \geq n_0$, $T(n) \leq Cn$

$$T(n) \leq T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right) + n, \quad \text{se } n > 10$$
$$\text{e } T(n) = 1, \quad \text{se } 1 \leq n \leq 10$$

- Hipótese de Indução: existe $\boldsymbol{C}$ positivo tal que $T(n) \leq \boldsymbol{C}n$
- Caso Base: $T(n) = 1 \leq \boldsymbol{C}n$ se $1 \leq n \leq 10$
- Passo indutivo:

  Assumimos que a hipótese é verdadeira para todo o $1 \leq n < k$, para $k > 10$

  $$T(k) \leq k + T\left(\frac{k}{5}\right) + T\left(\frac{7k}{10}\right)$$
  $$\leq k + \boldsymbol{C} \cdot \left(\frac{k}{5}\right) + \boldsymbol{C} \cdot \left(\frac{7k}{10}\right)$$
  $$= k + \frac{\boldsymbol{C}}{5}k + \frac{7\boldsymbol{C}}{10}k = \frac{\boldsymbol{19C}}{10}k \leq \boldsymbol{C}k$$
- Conclusão:

  Existe pelo menos um $\boldsymbol{C}$ tal que $\forall n \geq 1, T(n) \leq \boldsymbol{C}n$. Pela definição de limite superior, $T(n) = O(n)$.

- O **teorema principal** apresenta uma forma rápida para analisar a eficiência de algoritmos do tipo *divide-and-conquer*.

- Quando não é possível usar, devemos usar o método de indução (se necessário, com a técnica da substituição).

Vamos voltar a encontrar vários outros exemplos de algoritmos deste tipo em DAA.

Para reter das aulas anteriores

- O **teorema principal** apresenta uma forma rápida para analisar a eficiência de algoritmos do tipo *divide-and-conquer*
- Quando não é possível usar, devemos usar o método da indução e substituição
- Vamos tornar encontrar vários outros exemplos de algoritmos deste tipo em DAA.

Para estudar

- **4.3, 4.4, 4.5 – *Introduction to Algorithms*. Cormen, Leiserson et al., 3rd ed. MIT Press, 2009**

Para estudar

- **Cap. 2 e 3 – *Introduction to Algorithms.*** Cormen, Leiserson et al., 3rd ed. MIT Press, 2009
- **Cap. 2 – Algorithm *Design***, John Kleinberg, Eva Tàrdos, Addison-Wesley, 2005.