Estruturas de Dados e Básicas I - DIM0119

Selan R. dos Santos

DIMAp – Departamento de Informática e Matemática Aplicada
Sala 231, ramal 231, selan@dimap.ufrn.br
UFRN

2018.1

Complexidade e Busca – Conteúdo

Motivação e Objetivos

- **Motivação**
  - Para comparar algoritmos via análise matemática é necessário algum conhecimento sobre comportamento de funções.
  - A operação de **busca** é uma das mais importantes em Computação, visto que recupera dados para serem processados por algoritmos projetador para resolver problemas computacionais.
- **Objetivos**
  - Definir formalmente notação para complexidade matemática de algoritmos.
  - Determinar um método para comparar algoritmos que resolvem um mesmo problema computacional.
  - Apresentar técnicas elementares de busca de dados.

Crescimento de Funções

- Em geral os algoritmos que iremos estudar possuem tempo de execução proporcional a uma das funções abaixo:
  - 1: maior parte das instruções são executadas apenas uma ou algumas vezes, independente de n—tempo de execução constante.
  - $\log n$: ocorre em programas que resolve um problema maior transformado-o em uma série de subproblemas menores, assim reduzindo o tamanho do problema por uma certa constante fracionária a cada passo—tempo de execução logarítmico. Sempre que n dobra, $\log n$ aumenta de uma certa constante, mas $\log n$ não dobra até que n tenha sido aumentado para n^2. Ex.: $\log 1 = 0$, $\log 2 = 1$, $\log 4 = 2$, $\log 8 = 3, \ldots$
  - n: acontece quando uma pequena quantidade de processamento deve ser feito para cada elemento da entrada—tempo de execução linear. Esta é a situação *ótima* para um algoritmo que deve processar n entradas (ou gerar n saídas).

Crescimento de Funções

- ▷ $n \log n$: ocorre em algoritmos que quebra o problema principal em subproblemas menores, resolvendo-os e combinando as soluções—tempo de execução *"superlinear"* ou $n \log n$. Quando n dobra o tempo de execução torna-se um pouco mais do que o dobro.
- ⋆ *Os algoritmos, cujas complexidades sejam representadas pelas função apresentadas até aqui, são considerados como relativamente eficientes*
- ▷ n^2: tipicamente representa algoritmos que processa todos os pares de itens de dados—tempo de execução quadrático. Este tipo de complexidade é aceitável apenas para problemas relativamente pequenos. Quando n dobra o tempo de execução aumenta 4 vezes.
- ▷ n^3: similarmente refere-se a algoritmos que processam todos os (combinações de) triplas de itens de dados—tempo de execução cúbico. Quando n dobra o tempo de execução aumenta 8 vezes.
- ▷ 2^n: corresponde a algoritmos que utilizam força-bruta na solução de problemas—tempo de execução exponencial. Algoritmos com esta performance são impraticáveis. Sempre que n aumenta em 1 unidade o tempo de execução dobra. Ou então, quando n dobra o tempo de execução é elevado ao quadrado!

Complexidade de Tempo

Exemplo de Tempo Necessário para Solução de Problemas

- ▷ Veja abaixo a relação entre tempo de execução e tamanho de n, considerando que é necessário 1 segundo para processar cada item de dado

Complexidade	16 itens	32 itens	64 itens	128 itens
$O(\log_2(n))$	4 seg.	5 seg.	6 seg.	7 seg.
$O(n)$	16 seg.	32 seg.	64 seg.	128 seg.
$O(nlog_2(n))$	64 seg.	160 seg.	384 seg.	896 seg. (≈ 14.9 min.)
$O(n^2)$	256 seg.	17 min.	68 min.	273 min. (≈ 4h30min.)
$O(n^3)$	68 min.	546 min.	73 hor.	24 dias
$O(2^n)$	18 hor.	136 anos	500.000 milênios	???

- ▷ É possível utilizar *software* gráficos, como o `gnuplot`, para traçar gráficos das funções anteriores para diversos valores de n, de forma a poder comparar o crescimento do tempo de execução.

Complexidade de Tempo

Pior, Melhor e Casos Médio

- ▷ Seja A um algoritmo, $\{E_1, \ldots, E_m\}$ o conjunto de todas as entradas possíveis de A. Denote por t_i, o número de passos efetuados por A, quando a entrada for E_i. Definem-se:
  - ⋆ complexidade do pior caso $= max(E_i \in E\{t_i\})$,
  - ⋆ complexidade do melhor caso $= min(E_i \in E\{t_i\})$, e
  - ⋆ complexidade do caso médio $= \sum_{1 \leq i \leq m} p_i t_i$, onde p_i é a probabilidade de ocorrência da entrada E_i.
- ▷ A complexidade do pior caso corresponde ao número de passos que o algoritmo efetua para a entrada mais desfavorável. É uma das mais importantes (porque?) *Fornece um limite superior para o número de passos que o algoritmo pode efetuar, em qualquer caso*.

Limite Superior

Definição

- ▷ A notação mais comumente usada representar a complexidade de algoritmos é O (*big-Oh*), que permite expressar seu limite superior

Definição O (*big-Oh*) — limite superior

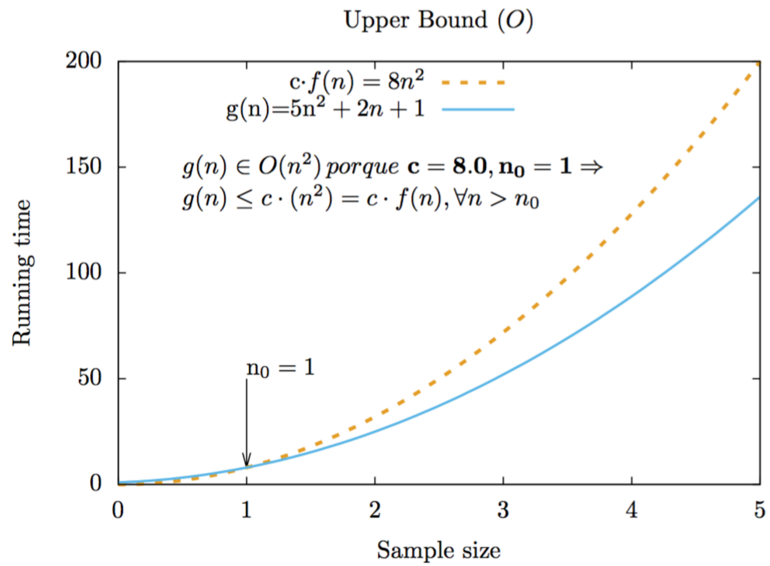
Sejam f e g funções reais positivas de variável inteira n.
A função $g(n) \in O(f(n))$, $g = O(f)$, se $\exists\ c_0 \in \mathbb{R} : c_0 > 0$ e $\exists\ n_0 \in \mathbb{I}$:

$$g(n) \leq c_0 \cdot f(n)$$

quando $n > n_0$.

- ▷ A partir de n_0, a função g fica sempre menor que a função f, com o fator multiplicativo c_0.
- ▷ Por exemplo, o algoritmo `máximo` (Exemplo 2) é $O(n)$.
- ▷ Uma fórmula com um termo O é denominada de expressão assintótica.

Limite Superior



Qual a Utilidade Desta Notação?

- O objetivo destas definições é estabelecer uma ordem relativa entre funções.
- Portanto estamos interessados nas taxas relativas de crescimento.
- Por exemplo:
  - Comparando $1000n$ e n^2. Quem cresce a uma taxa maior? Qual é o ponto de virada? $n = 1000$.
  - A definição diz que eventualmente existe um ponto n_0 após o qual $c_0 f(n)$ é sempre, pelo menos, tão alta quanto $g(n)$, de forma que, se as constantes podem ser ignoradas, $f(n)$ é pelo menos tão grande quanto $g(n)$.
  - No exemplo $g(n) = 1000n$, $f(n) = n^2$, $n_0 = 1000$ e $c_0 = 1$.
  - Portanto $1000n = O(n^2)$.

Limite Inferior

Definição

- A notação Ω (*big Omega*) expressa um limite inferior da complexidade

Definição Ω — limite inferior

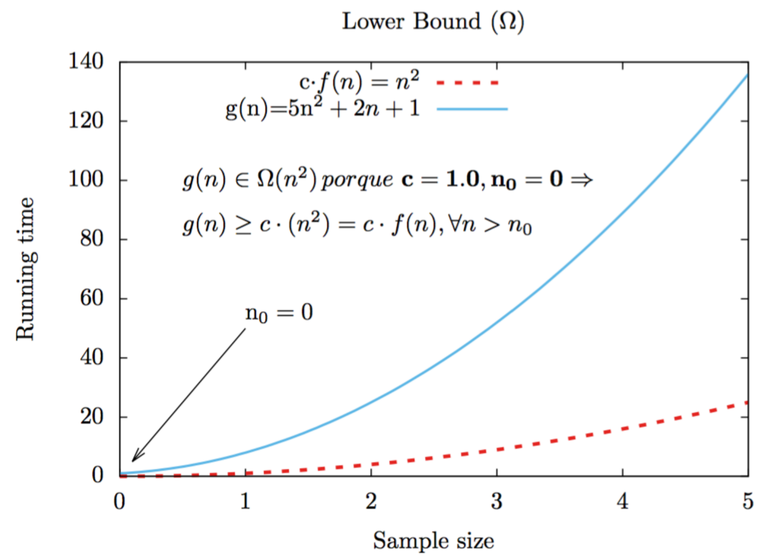
Sejam f e g funções reais positivas de variável inteira n.
A função $g(n) \in \Omega(f(n))$, $g = \Omega(f)$, se $\exists\ c_0 \in \mathbb{R} : c_0 > 0$ e $\exists\ n_0 \in \mathbb{I}$:

$$g(n) \geq c_0 . f(n)$$

quando $n > n_0$.

- A partir de n_0, a função g fica sempre maior que a função f, com o fator multiplicativo c_0.
- Problemas computacionais podem ter limite inferior: a ordenação com um único processador é $\Omega(n \cdot \log n)$. Portanto não há algoritmos sequenciais mais eficientes que $n \cdot \log n$ para ordenação.

Limite Inferior



Limite Justo
Definição

- ▷ A notação Θ (*big Theta*) expressa uma estimativa precisa da complexidade de um algoritmo.

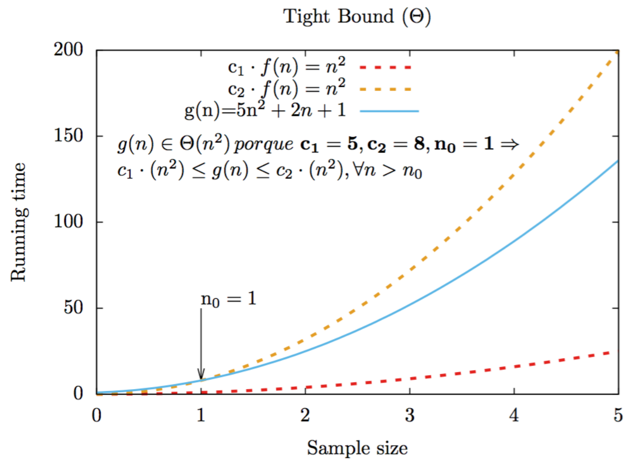
Definição Θ — limite justo

Sejam f e g funções reais positivas de variável inteira n.
A função $g(n) \in \Theta(f(n))$, $g = \Theta(f)$, se $g(n)$ for $O(f)$ e $\Omega(f)$ simultaneamente, ou seja, $\exists\ c_1, c_2 \in \mathbb{R} : c_1 > 0, c_2 > 0$ e $\exists\ n_0 \in \mathbb{I}$:

$$c_1.f(n) \leq g(n) \leq c_2.f(n)$$

quando $n > n_0$.

Limite Justo



Algoritmo Ótimo

Definição

Seja P um problema. Um limite inferior para P é uma função h, tal que a complexidade do pior caso de qualquer algoritmo que resolva P é $\Omega(h)$. Assim:

1. Todo algoritmo que resolve P efetua, pelo menos, $\Omega(h)$ passos;
2. Se existir um algoritmo A, cuja complexidade seja $O(h)$, então A é denominado de **algoritmo ótimo** para P;
3. Neste caso, o limite $\Omega(h)$ é o melhor (maior) possível.

Algoritmo Ótimo

- ▷ Considere o seguinte algoritmo para inverter um arranjo

```
procedimento inverte(V: arranjo de ref inteiro)
    var i, aux: inteiro                          #variáveis auxiliares
    var n: inteiro ←tam V                        #recupera o tamanho de V
    para i ← 0 até ⌊n/2⌋ − 1 faça                #percorrer vetor até antes do meio
        aux ← V[i]                               #salva o elemento da frente
        V[i] ← V[n − i − 1]                      #copia o do fim para frente
        V[n − i − 1] ← aux                       #copia o da frente para o fim
    fimpara
fim
```

- ▷ A notação $\lfloor x \rfloor$ significa piso de x e representa o maior inteiro $\leq x$; analogamente $\lceil x \rceil$ significa teto de x e corresponde ao menor inteiro $\geq x$. $\lfloor 9/2 \rfloor = 4$ e $\lceil 9/2 \rceil = 5$.
- ▷ Todo algoritmo tem que efetuar a leitura dos dados $\Rightarrow$ o problema de inversão de sequência **é** $\Omega(n)$. Como a complexidade do algoritmo é $O(n)$, conclui-se que trata-se de um algoritmo ótimo.

Regras Gerais para Análise de Algoritmos

Regra #1: Laços `For`

O tempo de execução para um laço `For` é, no máximo, o tempo de execução dos comandos internos do laço (incluindo os testes) vezes o número de iterações

Regra #2: Laços aninhados

Analise laços aninhados de dentro pra fora. O tempo de execução total de um comando dentro de um grupo de laços aninhados é o tempo de execução do comando multiplicado pelo produto do tamanho de todos os laços

- $t_1(n) * t_2(n) = O(f * g)$.

Regras Gerais para Análise de Algoritmos

Continuação

Por exemplo, o seguinte fragmento é $O(n^2)$:

```
para i ← 1 até N faça
    para j ← 1 até N faça
        Temp ← Temp + 1
    fimpara
fimpara
```

Regra #3: Comandos consecutivos

Comandos consecutivos simplesmente são adicionados. Isto na prática significa que o tempo total é o do comando com maior tempo, conforme item 1, já exposto:

Se $t_1(n) = O(f)$ e $t_2(n) = O(g)$ então

1. $t_1(n) + t_2(n) = O(f + g)$ (é $max(O(f), O(g))$)
2. $t_1(n) * t_2(n) = O(f * g)$.

Regras Gerais para Análise de Algoritmos

Continuação

Considere o fragmento abaixo, correspondente a um comando condicional composto:

```
se expressão então
    bloco S1
senão
    bloco S2
fim
```

Regra #4: Condicional Composto `Se/Senão`

Para um comando condicional composto o tempo total de execução não é nunca maior do que o tempo necessário para realizar o teste *mais* o maior dos tempos de execução entre os dois blocos do condicional (S1 e S2).

Eficiência Temporal de Algoritmos Não-recursivos

Plano Geral para Análise de Algoritmos Não-recursivos

- Decidir sobre o parâmetro n associado a entrada dos dados;
- Identificar a operação básica do algoritmo;
- Determinar as situações de entrada n que correspondem ao pior, melhor e médio casos;
- Montar o somatório (expressão) que representa o número de vezes que a operação básica é executada; e
- Simplificar o somatório com base em fórmulas padrão e regras[a].

[a] Consultar o Apêndice A do Levitin

Problema da Busca

Problema: busca sequencial em um vetor

Dado um conjunto de valores previamente armazenados em um vetor A, nas posições $A[l], A[l+1], \ldots, A[r]$, verificar se um número k está entre este conjunto de valores. Se o elemento procurado k não for encontrado a função deve retornar -1. Caso contrário deve retornar o índice do vetor A que contém o elemento k.

Desenvolver o algoritmo, calcular a complexidade para o melhor caso e pior caso para o algoritmo proposto.

Fórmula útil: $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$

Possíveis Soluções para o Problema da Busca

- Busca aleatória
  - ⋆ Se houver garantia de que um elemento não vai ser consultado mais de uma vez, a solução é alcançada em tempo finito.
  - ⋆ Se não houver garantia, pode-se não alcançar uma solução.
- Busca sequencial
  - ⋆ Mecanismo simples, que garante resposta em tempo finito e previsível.
  - ⋆ Analisar cada elemento do conjunto, desde o primeiro até:
    - localizar o elemento procurado ou
    - determinar que o elemento procurado não se encontra no conjunto porque: (1) percorreu todo o conjunto e não o encontrou, ou; (2) existe alguma informação extra sobre os dados que permite deduzirmos que o elemento procurado não está entre os elementos restantes.
- Outras alternativas?

Solução de busca sequencial em arranjo

Algoritmo: busca sequencial em um vetor

Entrada : Vetor A, chave k e limites de busca esquerdo l e direito r (inclusive).
Saída : Índice da 1ra ocorrência de k em A; ou -1, caso não exista k em A.
Precondição : $l \leq r$ e $l, r \geq 0$

```
função buscaSeq(A: arranjo de inteiro; k: inteiro; l: inteiro; r: inteiro): inteiro
    var i: inteiro                              # controlador do laço
    para i ← l até r faça                       # percorrer do esquerda (l) para direita (r)
        se k = A[i] então                       # achou o elemento procurado?
            retorna i                           # sim! então retornamos seu índice
    retorna −1                                  # caso não encontre, retorna índice inválido
```


Solução de busca sequencial em arranjo

Problema: busca sequencial em um vetor

Entrada : Vetor A, chave k e limites de busca esquerdo l e direito r (inclusive).
Saída : Índice da 1ra ocorrência de k em A; ou -1, caso não exista k em A.
Precondição : $l \leq r$ e $l, r \geq 0$

```
int buscaSeq( int A[ ], int k, int l, int , r ) {
    for ( int i = l; i <= r; ++i ) {
        if ( k == A[ i ] ) {
            return i;
        }
    return -1;
}
```

Solução de busca sequencial em arranjo
Análise de complexidade

Algoritmo: busca sequencial em um vetor

```
1: função buscaSeq(A: arranjo de inteiro; k: inteiro; l: inteiro; r: inteiro): inteiro
2:     var i: inteiro                                   # controlador do laço
3:     para i ← l até r faça                            # percorrer do esquerda (l) para direita (r)
4:         se k = A[i] então                            # achou o elemento procurado?
5:             retorna i                                # sim! então retornamos seu índice
6:     retorna −1                                       # caso não encontre, retorna índice inválido
```

- **Pior caso**: a chave k está localizada na última posição do vetor ou não está presente no vetor.
- $C_{pior}(n) = n$ (laço da linha 3 é repetido n vezes).

Solução de busca sequencial em arranjo
Análise de complexidade

Algoritmo: busca sequencial em um vetor

```
1: função buscaSeq(A: arranjo de inteiro; k: inteiro; l: inteiro; r: inteiro): inteiro
2:     var i: inteiro                                   # controlador do laço
3:     para i ← l até r faça                            # percorrer do esquerda (l) para direita (r)
4:         se k = A[i] então                            # achou o elemento procurado?
5:             retorna i                                # sim! então retornamos seu índice
6:     retorna −1                                       # caso não encontre, retorna índice inválido
```

- **Melhor caso**: a chave k está na primeira posição.
- $C_{melhor}(n) = c$, constante (linhas 3, 4 e 5).

Solução de busca sequencial recursiva em arranjo

- Como seria uma solução recursiva?

Algoritmo: busca sequencial recursiva em um vetor

Entrada : Vetor A, chave k e limites de busca esquerdo l e direito r (inclusive).
Saída : Índice da 1ra ocorrência de k em A; ou -1, caso não exista k em A.
Precondição : $l \leq r$ e $l, r \geq 0$

```
função buscaSeq(A: arranjo de inteiro; k: inteiro; l: inteiro; r: inteiro): inteiro
    var i: inteiro                                      # controlador do laço
    se l ≤ r então                                      # temos uma fixa ed busca válida?
        se k = A[l] então retorna l                     # elemento encontrado

        senão retorna buscaSeq(A, k, l + 1, r)          # recursão no resto do vetor

    senão
        retorna −1                                      # caso não encontre, retorna índice inválido
```


Problema da Busca (com restrição sobre os dados)

Problema: busca binária em um vetor ordenado

Dado um conjunto de valores previamente armazenados em um vetor A, nas posições $A[l], A[l+1], \ldots, A[r]$, **em ordem crescente**, verificar se um número v está entre este conjunto de valores. Se o elemento procurado v não for encontrado a função deve retornar -1. Caso contrário deve retornar o índice do vetor A que contém o elemento v.

Desenvolver o algoritmo, calcular a complexidade para o melhor caso e pior caso para o algoritmo proposto.

Solução de busca binária em arranjo

Princípio

- É possível reduzir a quantidade de comparações?
- Precisamos descartar o máximo possível de elementos após cada teste.
- Como fazemos para procurar uma palavra em um dicionário?
  - ⋆ Examina-se o nó do meio; se comparação não é positiva pode-se abandonar metade da lista.
  - ⋆ Aplicando-se esta ideia recursivamente até achar o elemento procurado ou esgotar a lista.
- Seja $A = \{a_0, a_1, \ldots, a_{n-1}\}$ um arranjo ordenado com n elementos tal que $a_i < a_j$ se $i < j$, para $i, j \in [0 \ldots n-1]$.
- Seja x o valor inteiro procurado em A:
  - ⋆ Se $n = 0$ então $x \notin A$ (A é $\emptyset$).
  - ⋆ Se $n > 0$ então:
    - Se $x = a_{\lfloor n/2 \rfloor}$, então $x \in A$ na posição $\lfloor n/2 \rfloor$.
    - Se $x < a_{\lfloor n/2 \rfloor}$, então aplicamos algoritmo a $\{a_0, \ldots, a_{\lfloor n/2 \rfloor - 1}\}$
    - Se $x > a_{\lfloor n/2 \rfloor}$, então aplicamos algoritmo a $\{a_{\lfloor n/2 \rfloor + 1}, \ldots, a_{n-1}\}$

Solução de busca binária em arranjo

Busca binária em um vetor (versão iterativa)

```
função buscaBin(A arranjo de inteiro; x: inteiro; l: inteiro; r: inteiro): inteiro
    var m: inteiro                                   # índice do meio
    enquanto r ≥ l faça                              # índices não cruzarem...
        m ← (l + r)/2                                # calcular a posição central do vetor
        se x = A[m] então                            # achou o elemento procurado?
            retorna m                                # sim! então retornamos seu índice
        senão se x < A[m] então                      # procurado está no subvetor esquerdo
            r ← m − 1                                # calcular novo limite da direita
        senão
            l ← m + 1                                # calcular novo limite da esquerda
        fim
    fim
    retorna −1                                       # caso não encontre, retorna índice inválido
fim
```


Análise de Complexidade da Busca Binária

- Melhor caso: o elemento procurado está no meio do vetor $\Rightarrow O(1)$.
- Pior caso: ocorre quando a chave procurada é o último elemento de L a ser testado ou então não está presente: lista é reduzida a 1 elemento
  - 1ª iteração: dimensão da lista é n,
  - 2ª iteração: dimensão da lista é $\lfloor n/2 \rfloor = \frac{n}{2}$,
  - 3ª iteração: dimensão da lista é $\lfloor (\lfloor n/2 \rfloor)/2 \rfloor = \frac{n}{2^2}$,
  - ...
  - mª iteração: dimensão da lista é $\frac{n}{2^m} = 1$.
- Portanto temos a sequência $n, \frac{n}{2}, \frac{n}{2^2}, \ldots, \frac{n}{2^m}$ e queremos saber o valor de m (quantidade de passos).
- Porém, sabemos que o último termo da sequência, $\frac{n}{2^m}$, é 1, ' logo temos que $m = \lg n$.
- Assim, no máximo o número de iterações é $1 + \lfloor \log_2 n \rfloor \Rightarrow O(\lg n)$.

Solução de busca binária em arranjo

Busca binária em um vetor (versão recursiva) em C++

```cpp
int nTentativas = 0;

int buscaBin( int A[], int x, int l, int r ) {
    int m = 0;
    if ( l > r )  return -1;
    else {
        m = ( l + r )/2;   // Calcular indice do meio.
        if ( x == A[ m ] )      return m;
        else if ( x < A[ m ] )  return buscaBin( A, x, l, m-1 );
        else                    return buscaBin( A, x, m+1, r );
```

Exercício Proposto

Problema: multiplicação de matrizes

Dados duas matrizes $n \times n$, A e B, desenvolver o algoritmo para computar o produto $C = AB$ e calcular a complexidade temporal.

Lembre que os elementos de $C_{i,j}$ são calculados através do produto escalar das linhas i de A com as colunas j de B.

Problema da multiplicação de matrizes

Solução

- ▷ Para tornar a solução mais genérica, vamos assumir que desejamos multiplicar $A_{n\times o} \cdot B_{o\times m} = C_{n\times m}$.
- ▷ Claro, se $n = o = m$ temos multiplicação de matrizes quadradas.

Problema: multiplicação de matrizes $n \times o$ por $o \times m$

procedimento `multiMat`(A: **arranjo de** $n \times o$ **inteiro**; B: **arranjo de** $o \times m$ **inteiro**, C: **arranjo de** $n \times m$ **ref inteiro**)

- **var** i, j, k: **inteiro** `# controle dos laços.`
- **para** $i \leftarrow 0$ **até** $n-1$ **faça** `# linhas da matriz resultado.`
  - **para** $j \leftarrow 0$ **até** $m-1$ **faça** `# colunas da matriz resultado.`
    - $C[i,j] \leftarrow 0$ `# inicializando resultado.`
    - **para** $k \leftarrow 0$ **até** $o-1$ **faça** `# acumular multiplicações.`
      - $C[i,j] \leftarrow A[i,k] * B[k,j] + C[i,j]$

Problema da multiplicação de matrizes

Análise

- ▷ A operação dominante é $C[i,j] \leftarrow A[i,k] * B[k,j] + C[i,j]$, com tempo c_1.
- ▷ Denominamos o tempo do laço mais interno de $L_1(o)$; o tempo do segundo laço de $L_2(m)$ e o tempo do laço mais externo de $L_3(n)$.
- ▷ O tempo geral é $T(n,m,o) \leq L_3(n)$, sendo que $L_3(n) = \sum_{i=0}^{n-1}(L_2(m))$, $L_2(m) = \sum_{j=0}^{m-1}(L_1(o))$ e $L_1(o) = \sum_{k=0}^{o-1} c_1$.
- ▷ Substituindo, temos $T(n,m,o) \leq \sum_{i=0}^{n-1}(\sum_{j=0}^{m-1}(\sum_{k=0}^{o-1} c_1))$.
- ▷ Ora, $\sum_{k=0}^{o-1} c_1 = c_1 \sum_{k=0}^{o-1} 1 = c_1 \cdot o$.
- ▷ Assim, $T(n,m,o) \leq \sum_{i=0}^{n-1}(\sum_{j=0}^{m-1}(c_1 \cdot o))$.
- ▷ Similarmente, $\sum_{j=0}^{m-1}(c_1 \cdot o) = (c_1 \cdot o) \cdot \sum_{j=0}^{m-1} 1 = (c_1 \cdot o) \cdot m$.
- ▷ Atualizando, $T(n,m,o) \leq \sum_{i=0}^{n-1}((c_1 \cdot o) \cdot m)$.
- ▷ Por fim, $\sum_{i=0}^{n-1}((c_1 \cdot o) \cdot m) = ((c_1 \cdot o) \cdot m) \cdot \sum_{i=0}^{n-1} 1 = ((c_1 \cdot o) \cdot m) \cdot n$
- ▷ Logo $T(n,m,o) \leq c_1 \cdot o \cdot m \cdot n$; se $n = m = o$, então $T(n) \leq c_1 \cdot n^3$.

Resumo do que é Necessário Saber

- ▷ Conceito de comportamento assintótico de função.
- ▷ Definição das notações de comparação de comportamento assintótico.
- ▷ Aplicação dessas notações para documentar a complexidade de algoritmos.
- ▷ Como determinar a complexidade de algoritmos (matemática ou experimental).
- ▷ Há problemas para os quais existe uma complexidade inerente.

Pontos Principais

- Tanto a eficiência temporal como espacial de um algoritmo são medidas em função do tamanho n da entrada fornecida ao algoritmo.
- Eficiência temporal é medida contando-se o número de vezes que a operação básica (dominante) é executada; a Eficiência espacial é medida contando-se o número de unidades de memória extras consumidas pelo algoritmo.
- A eficiência de alguns algoritmos pode diferir consideravelmente para entradas de mesmo tamanho; Neste caso, precisamos determinar o comportamento no pior, melhor e médio casos.
- O sistema de análise de algoritmos apresentado se preocupa primariamente com a ordem de crescimento do tempo de execução (e uso de memória) na medida em que a entrada cresce e se aproxima de um valor n muito grande.

Referências

J. Szwarcfiter and L. Markenzon.
Estruturas de Dados e Seus Algoritmos, 2ª edição, **Cap. 1**.
Editora LTC, 1994.

M. A. Weiss.
Data Structures and Algorithm Analysis in C++, 3rd edition. **Cap. 2**
Addision Wesley, 2006.

A. Levitin.
Introduction to the Design and Analysis of Algorithms, 3rd edition. **Cap. 2**
Addision Wesley, 2011.

D. Deharbe
Slides de Aula. **aula 2**
DIMAp, UFRN, 2006.