MAT A40 - Estrutura de Dados e Algoritmos I

Dr. George Lima
Departamento de Ciência da Computação
Instituto de Matemática e Estatística
Universidade Federal da Bahia

Módulo 1

Conceitos introdutórios e noções de complexidade de algoritmos

O que são algoritmos?

- Sequência não ambígua de instruções que, quando executada, produz resultados de saída a partir de sua entrada.
  - Por não-ambiguidade entende-se que as instruções podem ser executadas sem se fazer uso de inteligência.
  - Ausência de inteligência implica que algoritmos pode ser executados por máquinas.

Aspectos importantes para elaboração de algoritmos?

- Correção
  - Qualquer algoritmo nunca deve deixar de produzir resultados quando executados
  - Qualquer resultado produzido por um algoritmo deve estar de acordo com o esperado
  - Mas, como garantir a correção de algoritmos?
- Desempenho
  - Qualquer algoritmo deve ser elaborado para ser eficiente quanto ao uso dos recursos necessários à solução do problema para o qual ele foi especificado
  - Mas, como medir o desempenho de algoritmos?

Correção de algoritmos

- Garantia de correção: Em geral, não é possível correção para qualquer algoritmo
  - O número de possíveis entradas pode ser muito elevado para se testar todas elas
  - Em geral, não existe solução algorítmica para demonstrar que algoritmos estão corretos (vide problema da parada)
  - Mas, pode-se demonstrar correção, para casos ou algoritmos específicos executando sob condições específicas

Desempenho de algoritmos

- Medir desempenho: quanto mais recursos computacionais a serem usados, menos eficiente é o algoritmo
  - Não se pode confundir desempenho de algoritmo com desempenho de programa (sua implementação)
  - Fatores relacionados à infra-estrutura de execução, que envolve linguagem de programação, SO, hardware, não podem ser contabilizados como relacionados ao algoritmo

Exemplo de algoritmo

Busca sequencial:

- Entrada: sequência de n números inteiros $x_0, x_1, \ldots, x_{n-1}$ e um inteiro k
- Saída: o valor i tal que $x_i = k$ ou -1 caso não exista i tal que $x_i = k$.
- Algoritmo:
  1. Faça $i = 0$
  2. Se $x_i \neq k$ e $i < n$, execute os passos 2.1 e 2.2
     - 2.1 Incremente i de uma unidade
     - 2.2 Retorne ao passo 2
  3. Se $i = n$, devolva o valor -1 como saída. Caso contrário, devolva i como saída

Exemplo de algoritmo codificado em C

Busca sequencial:

```
1  /*
2  Entrada: vetor v, tamanho do vetor n, e um inteiro k.
3  Saida: indice do vetor que armazena k ou -1 se v nao
       contem k.
4   */
5   int seqSearch(int v[], int n, int k) {
6       int i = 0;
7
8     while (v[i] != k && i < n) i++;
9     if (i == n) return -1;
10    else return i;
11 }
```

Um algoritmo e seu programa correspondente

- O desempenho de um algoritmo não deve depender da infra-estrutura de execução. Portanto, medir os recursos computacionais usados executando o programa, não é recomendável!
  - Qual(is) entrada(as)?
  - Qual(is) cenários?
  - Qual infra-estrutura de execução?
- Recursos computacionais típicos
  - Processador (tempo de processamento)
  - Memória (espaço de armazenamento)
  - Número ou tamanho das mensagens transmitidas ou (banda da rede de comunicação)

Tempo de execução como métrica de desempenho

Dificuldades:

- Para medir o tempo de execução, temos que escolher uma entrada
- Ao medir o tempo de execução, estamos avaliando a implementação do algoritmo
- São várias as possíveis fontes de interferências durante as medições
- O resultado das medições não fornece garantias
- O resultado das medições não fornece representatividade de desempenho

Tempo de execução como métrica de desempenho

Primeiras ideias

1. Representatividade: pensar na pior entrada (pior caso)
   - Mas, qual seria o tamanho da entrada?
2. Removendo efeito de interferências: contar as instruções executadas
   - Mas, qual o custo de execução para cada instrução?

Tempo de execução como métrica de desempenho

Aplicando as primeiras ideias: assuma que existe um custo máximo c_j associado a cada passo j.

1. Faça $i = 0$ {custo c_1}
2. Se $x_i \neq k$ e $i < n$, execute os passos 2.1 e 2.2 {custo c_2}
   - 2.1 Incremente i de uma unidade {custo c_3}
   - 2.2 Retorne ao passo 2 {custo c_4}
3. Se $i = n$, devolva o valor de -1 como saída. Caso contrário, devolva i como saída {custo c_5}

Somando todos os custos para uma busca num vetor de tamanho n, o tempo de execução (pior caso) pode ser expresso por:

$$\begin{aligned} T(n) &= c_1 + c_2 \times (n+1) + (c_3 + c_4) \times n + c_5 \\ &= (c_2 + c_3 + c_4) \times n + c_1 + c_2 + c_5 + \\ &= An + B \qquad \therefore A = c_2 + c_3 + c_4, B = c_1 + c_2 + c_5 \end{aligned}$$

Tempo de execução como métrica de desempenho

Da função resultante da análise, sabe-se:

$$T(n) = An + B$$

- As constantes A e B são dependentes da infra-estrutura de execução. Portanto, elas não medem a eficiência do algoritmo e sim a da implementação do mesmo.
- Porém, independentemente das constantes A e B, sabe-se que o algoritmo piora seu tempo de execução linearmente em n. Seja onde este algoritmo for codificado (máquina de execução sequencial – von Newman), esta característica será mantida.
- É preciso uma métrica que desconsidere constantes.

Expressando desempenho e negligenciando constantes

Definição: Grande O

Uma função $f(n) \in O(g(n))$ se existem constantes $c > 0$, $n \geq 0$, tal que para todo $n \geq n_0, f(n) \leq cg(n)$.
Comumente, escreve-se $f(n) = O(g(n))$ ao invés de $f(n) \in O(g(n))$, mesmo sabendo-se que $O(g(n))$ é uma classe de funções.

Exemplo:

- Se $f(n) = An + B$, $f(n) = O(n)$, pois para que $f(n) \leq c\,n$,

$$An + B \leq c\,n \Rightarrow A + \frac{B}{n} \leq c,$$

se $n > 0$. Portanto, fazendo $c = A + B$, a relação acima é verdade para todo $n \geq 1$.

Expressando desempenho e negligenciando constantes

Falso ou verdade?

1. $An + B = O(n^2)$
2. $105n^2 + 23n + 109 = O(n^2)$
3. $log_2 n = O(n)$
4. $log_2 n = O(\sqrt{n})$
5. $10^{10} n + 100^{10} = O(n)$
6. $\sum_{i=1}^{n} \sum_{j=i}^{n} j = O(n^2)$
7. $\sum_{i=1}^{n} O(i) = O(n)$
8. $n^4 + O(n^5) + 345n = O(n^5)$
9. $(\log n)^a = O(n^b)$, para qualquer b e qualquer a positivo.
10. $f(n) + g(n) = O(\max\{f(n) + g(n)\})$

Tempo de execução como métrica de desempenho

Qual o desempenho do seguinte trecho, em notação O?
[extraído de Al Aho and Jeff Ullman (1994), capítulo 3]

```
1  for (i = 0; i < n-1; i++) {
2    small = i;
3    for (j = i+1; j < n; j++)
4      if (A[j] < A[small])
5        small = j;
6
7      temp = A[small];
8      A[small] = A[i];
9      A[i] = temp;
10 }
```

Tempo de execução como métrica de desempenho

Qual o desempenho do seguinte trecho, em notação O?
[extraído de Al Aho and Jeff Ullman (1994), capítulo 3]

```
1  for (i = 0; i < n-1; i++) {
2    small = i;
3    for (j = i+1; j < n; j++)
4      if (A[j] < A[small])
5        small = j;
6
7      temp = A[small];
8      A[small] = A[i];
9      A[i] = temp;
10 }
```

Podemos dizer que $T(n) = O(n^3)$ ou $T(n) = O(2^n)$?

Limites assintóticos superior, inferior e exato

Definição: Grande O

Uma função $f(n) \in O(g(n))$ se existem constantes $c > 0$, $n \geq 0$, tal que para todo $n \geq n_0, f(n) \leq cg(n)$.

Definição: Grande Ω

Uma função $f(n) \in \Omega(g(n))$ se existem constantes $c > 0$, $n \geq 0$, tal que para todo $n \geq n_0, f(n) \geq cg(n)$.

Definição: Θ

Uma função $f(n) \in \Theta(g(n))$ sse $f(n) \in \Omega(g(n))$ e $f(n) \in O(g(n))$.

Análise de complexidade

Exemplo: fatores de 2
[extraído de Al Aho and Jeff Ullman (1994), capítulo 3]

```
1
2  int PowersOfTwo(int n) {
3    int i = 0;
4
5    while (n%2 == 0) {
6      n = n/2;
7      i++;
8    }
9  return i;
10 }
```

Análise de complexidade

Exemplo: fatores de 2
[extraído de Al Aho and Jeff Ullman (1994), capítulo 3]

```
1
2  int PowersOfTwo(int n) {
3    int i = 0;
4
5    while (n%2 == 0) {
6      n = n/2;
7      i++;
8    }
9  return i;
10 }
```

A função exata do tempo de execução é complexa. O pior caso restringe-se a quando n é potência de 2: $T(n) = \Theta(\log n)$.

Análise de complexidade

Exemplo: pesquisa binária

```
1  int binSearch(int v[], int k, int n) {
2
3    int i;
4    int b = n;
5    int a = 0;
6
7    while (a <= b) {
8      i = (int) (a+b)/2;
9      if (v[i] == k) return (i);
10     else if (v[i] > k) b = i-1;
11     else a = i+1;
12   }
13   return(-1);
14 }
```

Análise de complexidade

Exemplo: pesquisa binária

```
1  int binSearch(int v[], int k, int n) {
2
3    int i;
4    int b = n;
5    int a = 0;
6
7    while (a <= b) {
8      i = (int) (a+b)/2;
9      if (v[i] == k) return (i);
10     else if (v[i] > k) b = i-1;
11     else a = i+1;
12   }
13   return(-1);
14 }
```

Contar o número máximo de iterações (pior caso):
$T(n) = \Theta(\log n)$

Análise de complexidade

Exemplo: teste de primalidade [extraído de Al Aho and Jeff Ullman (1994), capítulo 3]

```
1 int prime(int n) {
2   int i = 2;
3   while (i*i <= n)
4     if (n%i == 0) return FALSE;
5     else i++;
6
7   return TRUE;
8 }
```

Análise de complexidade

Exemplo: teste de primalidade [extraído de Al Aho and Jeff Ullman (1994), capítulo 3]

```
1 int prime(int n) {
2   int i = 2;
3   while (i*i <= n)
4     if (n%i == 0) return FALSE;
5     else i++;
6
7   return TRUE;
8 }
```

Se o número n é primo (pior caso), não há mais que $\lceil \sqrt{n} \rceil$ iterações: $T(n) = \Theta(\sqrt{n})$

Análise de complexidade

Exemplo: teste de primalidade [extraído de Al Aho and Jeff Ullman (1994), capítulo 3]

```
1 int prime(int n) {
2   int i = 2;
3   while (i*i <= n)
4     if (n%i == 0) return FALSE;
5     else i++;
6
7   return TRUE;
8 }
```

Se o número n é primo (pior caso), não há mais que $\lceil \sqrt{n} \rceil$ iterações: $T(n) = \Theta(\sqrt{n})$

Para pensar: o tempo de execução deste algoritmo é polinomial ou exponencial em relação à sua entrada?