decom
departamento
de computação

UFOP

Análise de Algoritmos

Parte 1

Prof. Túlio Toffolo
http://www.toffolo.com.br

BCC202 – Aula 04

Algoritmos e Estruturas de Dados I

Qual a diferença entre um algoritmo e um programa?

Como escolher o algoritmo mais adequado para uma situação?

Análise de Algoritmos

- Analisar um algoritmo consiste em "verificar" o que?
  - **Tempo de Execução**
  - **Espaço/memória ocupada**

- Esta análise é necessária para escolher o algoritmo mais adequado para resolver um dado problema.
- É especialmente importante em áreas como pesquisa operacional, otimização, teoria dos grafos, estatística, probabilidades, entre outras.

Cálculo do Custo

- **Cálculo do Custo pela Execução do Algoritmo**
  - Tal medidas é bastante **inadequada** e o resultado não pode ser generalizado;
  - Os resultados são dependentes do compilador, que pode favorecer algumas construções em detrimento de outras;
  - Os resultados dependem do *hardware*;
  - Quando grandes quantidades de memória são utilizadas, as medidas de tempo podem depender deste aspecto.

Cálculo do Custo

- **Cálculo do Custo pela Execução do Algoritmo**
  - Apesar disso, há argumentos a favor de se obterem medidas reais de tempo.
  - Ex.: quando há vários algoritmos distintos para resolver um mesmo tipo de problema, todos com um custo de execução dentro de uma mesma **ordem de grandeza**.

Análise de Algoritmos

- Possibilidade de analisar:
  - **Um algoritmo particular**.
  - **Uma classe de algoritmos.**

Análise de Algoritmos

- Análise de um **algoritmo particular**.
  - Qual é o custo de usar um dado algoritmo para resolver um problema específico?
  - Características que devem ser investigadas:
    - Análise do número de vezes que cada parte do algoritmo deve ser executada (**tempo**)
    - Estudo da quantidade de memória necessária (**espaço**).

Análise de Algoritmos

- Análise de uma **classe de algoritmos**.
  - Qual é o algoritmo de menor custo possível para resolver um problema particular?
  - Toda uma família de algoritmos é investigada.
  - Procura-se identificar um que seja o melhor possível.
  - Coloca-se **limites** para a complexidade computacional dos algoritmos pertencentes à classe.

Custo de um Algoritmo

- O menor custo possível para resolver problemas de uma classe nos dá a dificuldade inerente para resolver o problema.
- **Quando o custo de um algoritmo é igual ao menor custo possível, o algoritmo é _____ para a medida de custo considerada.**

  Podem existir vários algoritmos ótimos para resolver o mesmo problema.
- Se a mesma medida de custo é aplicada a diferentes algoritmos, então é possível compará-los e escolher o mais adequado.

Custo de um Algoritmo

- Utilizaremos um modelo matemático baseado em um computador idealizado.
- Deve ser especificado o conjunto de operações e seus custos de execuções.
- É mais usual ignorar o custo de algumas operações e considerar apenas outras mais significativas.
- Ex.: algoritmos de ordenação. Consideramos o **número de comparações** entre os elementos do conjunto a ser ordenado e ignoramos as demais operações.

Função de complexidade

- Para medir o custo de execução de um algoritmo vamos definir uma **função de complexidade** ou função de custo ***f***.
- Função de **complexidade de tempo**: *f(n)* mede o tempo necessário para executar um algoritmo em um problema de tamanho n.
- Função de **complexidade de espaço**: *s(n)* mede a memória necessária para executar um algoritmo em um problema de tamanho n.

Função de complexidade

- Utilizaremos ***f*** para denotar uma função de complexidade de tempo daqui para a frente.
- A complexidade de tempo na realidade não representa tempo diretamente
  - Representa o número de vezes que determinadas operações relevantes são executadas.

Exemplo: maior elemento

- Considere o algoritmo para encontrar o maior elemento de um vetor de inteiros A[n]; n ≥ 1.

```
int Max(int* A, int n) {
    int i, Temp;

    Temp = A[0];
    for (i = 1; i < n; i++)
        if (Temp < A[i])
            Temp = A[i];
    return Temp;
}
```

- Seja *f* uma função de complexidade tal que *f(n)* é o número de comparações envolvendo os elementos de A, se A contiver n elementos.
- **Qual a função f(n)?**

Exemplo: maior elemento

- **Teorema**: Qualquer algoritmo para encontrar o maior elemento de um conjunto com n elementos, n ≥ 1, faz pelo menos n -1 comparações.
- **Prova**: Cada um dos n - 1 elementos tem de ser investigado por meio de comparações, que é menor do que algum outro elemento.
  - Logo, **n-1 comparações são necessárias.**

O teorema acima nos diz que, se o número de comparações for utilizado como medida de custo, então a função Max do programa anterior é ótima.

Tamanho da entrada de dados

- A medida do custo de execução de um algoritmo **depende principalmente** do tamanho da entrada dos dados.
- É comum considerar o tempo de execução de um programa como uma função do tamanho da entrada.
- Para alguns algoritmos, o custo de execução é uma função da entrada particular dos dados, não apenas do tamanho da entrada.

Tamanho da entrada de dados

- No caso da função Max do programa do exemplo, o **custo é uniforme** sobre todos os problemas de tamanho n.
- Já para um **algoritmo de ordenação isso não ocorre**: se os dados de entrada já estiverem quase ordenados, então pode ser que o algoritmo trabalhe menos.

Melhor caso, pior caso e caso médio

- **Melhor caso**: menor tempo de execução sobre todas as entradas de tamanho n.
- **Pior caso**: maior tempo de execução sobre todas as entradas de tamanho n.
- **Caso médio** (ou caso esperado): média dos tempos de execução de todas as entradas de tamanho n.

Melhor caso <= Caso médio <= Pior caso

Análise do melhor caso, pior caso e caso médio

- Na análise do caso médio esperado, supõe-se uma **distribuição de probabilidades** sobre o conjunto de entradas de tamanho n e o custo médio é obtido com base nessa distribuição.
- A análise do caso médio é geralmente **muito mais difícil de se obter** do que as análises do melhor e do pior caso.

- É comum supor uma distribuição de probabilidades em que todas as entradas possíveis são **igualmente prováveis**.
- Na prática isso nem sempre é verdade.

Exemplo: registros de um arquivo

- Considere o problema de acessar os **registros** de um arquivo.
- Cada registro contém uma **chave** única que é utilizada para recuperar registros do arquivo.
- O problema: dada uma chave qualquer, localize o registro que contenha esta chave.
- O algoritmo de pesquisa mais simples é o que faz a **pesquisa sequencial**.

Exemplo: registros de um arquivo

- Seja f uma função de complexidade *f(n)*
- Seja f(n) o número de registros consultados no arquivo (número de vezes que a chave de consulta é comparada com a chave de cada registro).

- **Melhor caso:**
  - O registro procurado é o primeiro consultado !!!
  - *f(n) = 1*

Exemplo: registros de um arquivo

- **Pior caso:**
  - registro procurado é o último consultado ou não está presente no arquivo;
  - $f(n) = n$

- **Caso médio:**
  - O caso médio nem sempre é tão simples de calcular;
  - Como faremos neste problema???

Exemplo: registros de um arquivo

- No estudo do caso médio, vamos considerar que toda pesquisa recupera um registro.
- Se p_i for a probabilidade de que o i-ésimo registro seja procurado, e considerando que para recuperar o i-ésimo registro são necessárias *i* comparações, então:

$$f(n) = 1 \times p_1 + 2 \times p_2 + 3 \times p_3 + \ldots + n \times p_n$$

Exemplo: registros de um arquivo

- Para calcular f(n) basta conhecer a distribuição de probabilidades p_i.
- Se cada registro tiver a mesma probabilidade de ser acessado que todos os outros, então

$$p_i = 1/n,\ 1 \leq i \leq n$$

Exemplo: registros de um arquivo

- Para calcular f(n) basta conhecer a distribuição de probabilidades p_i.

$$p_i = 1/n,\ 1 \leq i \leq n$$

- A análise do caso esperado revela que uma pesquisa com sucesso examina aproximadamente metade dos registros.

$$f(n) = \frac{1}{n}(1+2+3+\cdots+n) = \frac{1}{n}\left(\frac{n(n+1)}{2}\right) = \frac{n+1}{2}.$$

Exemplo: registros de um arquivo

- **Melhor caso:**
  - O registro procurado é o primeiro consultado !!!
  - *f(n) = 1*
- **Pior caso:**
  - registro procurado é o último consultado ou não está presente no arquivo;
  - *f(n) = n*
- **Caso médio:**
  - *f(n) = (n+1)/2*

Exemplo: maior e menor elemento (1)

- Considere o problema de encontrar o maior e o menor elemento de um vetor de inteiros A[n]; n ≥ 1.
- Um algoritmo simples pode ser derivado do algoritmo apresentado no programa para achar o maior elemento.

```
void MaxMin1(int* A, int n, int* pMax, int* pMin) {
   int i;
   *pMax = A[0];
   *pMin = A[0];
   for (i = 1; i < n; i++) {
      if (A[i] > *pMax)
          *pMax = A[i];
      if (A[i] < *pMin)
          *pMin = A[i];
   }
}
```

Qual a função de complexidade?

```
void MaxMin1(int* A, int n, int* pMax, int* pMin) {
    int i;

    *pMax = A[0];
    *pMin = A[0];
    for (i = 1; i < n; i++) {
        if (A[i] > *pMax)
            *pMax = A[i];
        if (A[i] < *pMin)
            *pMin = A[i];
    }
}
```

2*(n-1)

Qual a função de complexidade?

- Seja *f(n)* o número de comparações entre os elementos de A, se A contiver *n* elementos.
- Logo ***f(n) = 2(n-1)*** para *n > 0*, para o melhor caso, pior caso e caso médio.

```
void MaxMin1(int* A, int n, int* pMax, int* pMin) {
   int i;
   *pMax = A[0];
   *pMin = A[0];
   for (i = 1; i < n; i++) {
      if (A[i] > *pMax)
          *pMax = A[i];
      if (A[i] < *pMin)
          *pMin = A[i];
   }
}
```

Exemplo: maior e menor elemento (2)

- MaxMin1 pode ser facilmente melhorado: a comparação A[i] < *pMin só é necessária quando a comparação A[i] > *pMax dá falso.

```
void MaxMin2(int* A, int n, int* pMax, int* pMin) {
   int i;
   *pMax = A[0];
   *pMin = A[0];
   for (i = 1; i < n; i++) {
      if (A[i] > *pMax)
          *pMax = A[i];
      else if (A[i] < *pMin)
          *pMin = A[i];
   }
}
```

Qual a função de complexidade?

Melhor caso:

- quando os elementos estão em ordem crescente;
- ***f(n) = n – 1***

Pior caso:

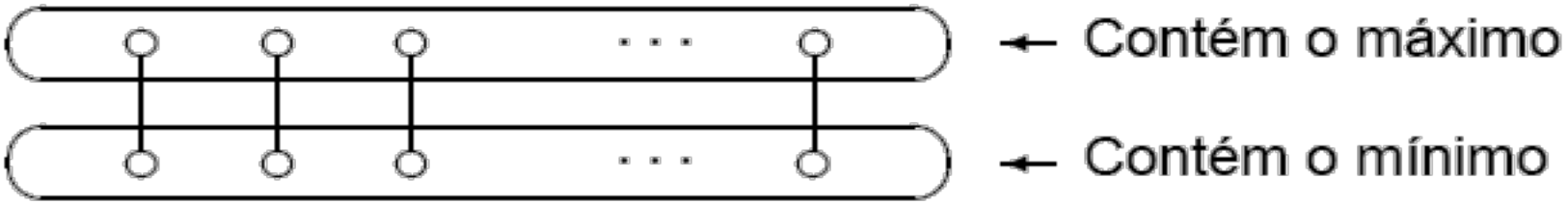
- quando o maior elemento é o primeiro no vetor;
- ***f(n) = 2(n – 1)***

Caso médio:

- No caso médio, A[i] é maior do que Max a metade das vezes.
- ***f(n) = 3n/2 – 3/2***

Exemplo: maior e menor elemento (3)

- Considerando o número de comparações realizadas, existe a possibilidade de obter um algoritmo mais eficiente:
  1. Compare os elementos de A aos pares, separando-os em dois subconjuntos (maiores em um e menores em outro), a um custo de ⌈n/2⌉ comparações.
  2. O máximo é obtido do subconjunto que contém os maiores elementos, a um custo de ⌈n/2⌉ -1 comparações
  3. O mínimo é obtido do subconjunto que contém os menores elementos, a um custo de ⌈n/2⌉ -1 comparações



Exemplo: maior e menor elemento (3)

- Os elementos de A são comparados dois a dois. Os elementos maiores são comparados com *p*Max* e os elementos menores são comparados com *p*Min*.
- Quando n é ímpar, o elemento que está na posição A[n-1] é duplicado na posição A[n] para evitar um tratamento de exceção.
- Para esta implementação:

no pior caso, melhor caso e caso médio

$$f(n) = \frac{n}{2} + \frac{n-2}{2} + \frac{n-2}{2} = \frac{3n}{2} - 2,$$

Exemplo: maior e menor elemento (3)

```
void MaxMin3(int* A, int n, int* pMax, int* pMin) {
   int i, FimDoAnel;

   if ((n % 2) > 0) { A[n] = A[n-1]; FimDoAnel = n; }
   else             { FimDoAnel = n-1; }

   if (A[0] > A[1]) { *pMax = A[0]; *pMin = A[1]; }   → Comparação 1
   else             { *pMax = A[1]; *pMin = A[0]; }

   for (i=2; i<FimDoAnel; i+=2) {
      if (A[i] > A[i+1]) {                            → Comparação 2
         if (A[i]   > *pMax) *pMax = A[i];            → Comparação 3
         if (A[i+1] < *pMin) *pMin = A[i+1];          → Comparação 4
      }
      else {
         if (A[i]   < *pMin) *pMin = A[i];            → Comparação 3
         if (A[i+1] > *pMax) *pMax = A[i+1];          → Comparação 4
      }
   }
}
```

Qual a função de complexidade?

- Quantas comparações são feitas em MaxMin3?
  - 1ª. comparação feita 1 vez
  - 2ª. comparação feita n/2 - 1 vezes
  - 3ª. e 4ª. comparações feitas n/2 – 1 vezes

$$f(n) = 1 + n/2 - 1 + 2 * (n/2 - 1)$$

$$f(n) = (3n - 6)/2 + 1$$

$$f(n) = 3n/2 - 3 + 1 = 3n/2 - 2$$

Comparação entre os algoritmos "MaxMin"

Os três algoritmos	$f(n)$		
	Melhor caso	Pior caso	Caso médio
MaxMin1	$2(n-1)$	$2(n-1)$	$2(n-1)$
MaxMin2	$n-1$	$2(n-1)$	$3n/2-3/2$
MaxMin3	$3n/2-2$	$3n/2-2$	$3n/2-2$

- A tabela apresenta uma comparação entre os algoritmos dos programas MaxMin1, MaxMin2 e MaxMin3, considerando o número de comparações como medida de complexidade.
- Os algoritmos MaxMin2 e MaxMin3 são superiores ao algoritmo MaxMin1.
- O algoritmo MaxMin3 é superior ao algoritmo MaxMin2 com relação ao pior caso e bastante próximo quanto ao caso médio.

Exemplo: exercício da primeira aula

- *f(n) = 7*

```
int nroNotas(int v[], int valor)
{
    v[0] = valor/50;
    valor = valor%50;
    v[1] = valor/10;
    valor = valor%10
    v[2] = valor/5;
    valor = valor%5;
    v[3] = valor;
}
```

Exemplo: exercício da primeira aula

- *f(n) = 4 + 4 + 3*NúmeroNotas*

```
int nroNotas(int v[], int valor) {
    v[0] = 0; v[1] = 0; v[2] = 0; v[3] = 0;
    while (valor >= 50) {
        v[0]++;
        valor = valor - 50;
    }
    while (valor >= 10) {
        v[1]++;
        valor = valor - 10;
    }
    while (valor >= 5) {
        v[2]++;
        valor = valor - 5;
    }
    while (valor >= 1) {
        v[3]++;
        valor = valor - 1;
    }
}
```

Perguntas?

O que é ter excelência?

É uma atitude, um hábito!

Quem vê, entende...

Quem faz, aprende!!!

Exercício

```
void exercicio1 (int n)
{
  int i, a;
  a = 0; i = 0;
  while (i < n) {
    a += i;
    i += 2;
  }
}
```

```
void exercicio2 (int n)
{
  int i, j, a;
  a = 0;
  for (i = 0; i < n; i++)
    for (j = 0; j < i; j++)
      a += i + j;
}
```