Análise de Complexidade

Estrutura de Dados II

Prof. João Dallyson Sousa de Almeida
Núcleo de Computação Aplicada NCA - UFMA
Dep. De Informática - Universidade Federal do Maranhão

Apresentação

- Ementa
  - Algoritmos de ordenação e busca.
  - Árvore de busca multidirecional balanceada.
  - Hashing. Noções de organização de arquivos.
  - Noções de grafos: conceitos, coloração, árvores geradoras..
  - Algoritmos em grafos: caminho mínimo, fluxo máximo e outros.

- Bibliografia: básica
  - CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L.; STEIN, C. Algoritmos: Teoria e Prática. Editora Campus, 2002
  - Algorithms 4th edition by R. Sedgewick and K. Wayne, Addison-Wesley Professional, 2011, ISBN 0-321-57351-X
  - Ziviani, N. Projeto de Algoritmos Com Implementações em Pascal e C, Cengage Learning, 2004.

- Bibliografia: complementar
  - TENENBAUM, Aaron; LANGSAM, Yedidyah; AUGENSTEIN, Moshe J. Estruturas de dados usando C. São Paulo: Makron Books, 1995. ISBN: 9788534603485
  - ASCENCIO, Ana Fernanda Gomes; ARAUJO, Graziela Santos. Estruturas de Dados: Algoritmos, análise da complexidade e implementações em Java e C/C++. Pearson Prentice Hall, 2010
  - DROZDEK, Adam. Adam Drozdek. Data Structures and Algorithms in Java. 2. Cengage Learning. 2004. 2. Cengage Learning. 2004
  - GOODRICH, Michael T. Estruturas de dados e algoritmos em java. 4 ED. Porto Alegre: Bookman, 2007. 600.
  - SKIENA, Steven S.. The Algorithm Design Manual. 2. Springer-Verlag. 2008

Cálculo da Complexidade: Algoritmo iterativo

```
void FloidWarshall(int dist[][]) {
        int i;                                          1 +
        int j;                                          1 +
        int k;                                          1 +
        for ( k = 0; k < n; k++ ) {                     n * (
                for ( i = 0; i < n; i++ ) {             n * (
                        for ( j = 0; j < n; j++ ) {     n * (
                                int a = dist[i][j];     1 +
                                int b = dist[j][k];     1 +
                                int c = dist[k][j];     1 +
                                dist[i][j] = min(a, b + c );   3
                        }                               )
                }                                       )
        }                                               )
}

int min(int a, int b) {
        if(a < b)                                       1 +
                return a;                               1 +
        return b;                                       1
}
```

FONTE: notas de aula prof. Rafael / IFMA

Cálculo da complexidade: Algoritmo iterativo

$$
\begin{aligned}
f(n) \\
&= 1 + 1 + 1 \\
&+ n\left(n(n(1 + 1 + 1 + 3))\right) \\
&= 3 + n(n(n(6))) = 6n^3 + 3
\end{aligned}
$$

$$\boxed{f(n) \in \boldsymbol{O}(\boldsymbol{n}^3)}$$

Cálculo Passo a passo

- **Divida** o algoritmo em **pedaços menores** e **analise** a complexidade de **cada um deles**.
  - Ex: analise o corpo de um laço de repetição e depois veja quantas vezes ele é executado;

- **Procure os laços de repetição** que operam sobre toda uma estrutura de dados. Se você sabe o tamanho da estrutura de dados, você sabe quantas vezes o laço é executado e consequentemente o tempo de execução do laço de repetição.

Exemplo

```
public int arrayMax(Vector V) {
    int currentMax = (Integer)V.get(0)).intValue;
    for(int i=1; i<V.size();i++) {
       if (((Integer)V.get(i)).intValue > currentMax)
        currentMax=((Integer)V.get(i)).intValue;
    }
    return (currentMax);
}
```

Custo 1

Total fora do loop : 3

Custo 1 por instrução

Número de repetições: N-1

Total : 4(N-1)

$$T(n) = 4*(n-1) + 3$$

Classes de Comportamento Assintótico

- Se f é uma função de complexidade para um algoritmo F, então O(f) é considerada a complexidade assintótica ou o comportamento assintótico do algoritmo F.
  - A relação de dominação assintótica permite comparar funções de complexidade.
  - Entretanto, se as funções f e g dominam assintoticamente uma a outra, então os algoritmos associados são equivalentes.
  - Nestes casos, o comportamento assintótico não serve para comparar os algoritmos

Exemplo

- Por exemplo, considere dois algoritmos F e G aplicados à mesma classe de problemas, sendo que F leva três vezes o tempo de G ao serem executados, isto é, $f(n) = 3g(n)$, sendo que $O(f(n)) = O(g(n))$.

- Logo, o comportamento assintótico não serve para comparar os algoritmos F e G, porque eles diferem apenas por uma constante

Comparação de Programas

- Podemos avaliar programas comparando as funções de complexidade, relaxando as constantes de proporcionalidade.

- Um programa com tempo de execução $O(n)$ é melhor que outro com tempo $O(n^2)$.

- Porém, as constantes de proporcionalidade podem alterar esta consideração.

Exemplo

- Um programa leva 100n unidades de tempo para ser executado e outro leva $2n^2$. Qual dos dois programas é melhor?

Principais Classes de Problemas

- $f(n) = O(1)$:
  - Algoritmos de complexidade O(1) são ditos de **complexidade constante.**
  - **Uso do algoritmo independe de n.**
  - **As instruções do algoritmo são executadas** um número fixo de vezes.

```
if (condição == true) then {
        realiza alguma operação em tempo constante
}
else  {
        realiza alguma operação em tempo constante
}
```

Principais Classes de Problemas (2)

- $f(n) = O(\log n)$.
  - Um algoritmo de complexidade $O(\log n)$ é dito de complexidade logarítmica.
  - Típico em algoritmos que transformam um problema em outros menores.
  - Pode-se considerar o tempo de execução como menor que uma constante grande.
  - Quando n é mil, $\log_2 n \approx 10$, quando n é 1 $milhão, $\log_2 n \approx 20$.
  - Para dobrar o valor de log n temos de considerar o quadrado de n.

Principais Classes de Problemas (3)

- $f(n) = O(n)$.
  - Um algoritmo de complexidade $O(n)$ é dito de complexidade linear.
  - Em geral, um pequeno trabalho é realizado sobre cada elemento de entrada.
  - É a melhor situação possível para um algoritmo que tem de processar/produzir n elementos de entrada/saída.
  - Cada vez que n dobra de tamanho, o tempo de execução também dobra.

Principais Classes de Problemas (3)

- $f(n) = O(n)$.

```
for (i = 0; i < N; i = i + 1 ) {
        if (condição == true) then {
                realiza alguma operação em tempo constante
        }
        else  {
                realiza alguma operação em tempo constante
        }
}
```

Principais Classes de Problemas (4)

- $f(n) = O(n \log n)$.
  - Típico em algoritmos que quebram um problema em outros menores, resolvem cada um deles independentemente e juntando as soluções depois.
  - Quando n é 1 milhão e a base do logaritmo é 2, $n\log_2 n$ é cerca de 20 milhões.
  - Quando n é 2 milhões e a base do logaritmo é 2, $n\log_2 n$ é cerca de 42 milhões, pouco mais do que o dobro.

Principais Classes de Problemas (5)

- $f(n) = O(n^2)$.
  - Um algoritmo de complexidade $O(n^2)$ é dito de complexidade quadrática.
  - Ocorrem quando os itens de dados são processados aos pares, muitas vezes em um anel dentro de outro.
  - Quando n é mil, o número de operações é da ordem de 1 milhão.
  - Sempre que n dobra, o tempo de execução é multiplicado por 4.
  - Úteis para resolver problemas de tamanhos relativamente pequenos.

Principais Classes de Problemas (6)

- $f(n) = O(n^3)$:
  - Um algoritmo de complexidade $O(n^3)$ é dito de complexidade cúbica.
  - Úteis apenas para resolver pequenos problemas.
  - Quando n é 100, o número de operações é da ordem de 1 milhão.
  - Sempre que n dobra, o tempo de execução fica multiplicado por 8.

Principais Classes de Problemas (7)

- $f(n) = O(2^n)$.
  - Um algoritmo de complexidade $O(2^n)$ é dito de complexidade exponencial.
  - Geralmente não são úteis sob o ponto de vista prático.
  - Ocorrem na solução de problemas quando se usa força bruta para resolvê-los.
  - Quando n é 20, o tempo de execução é cerca de 1 milhão. Quando n dobra, o tempo fica elevado ao quadrado.

Principais Classes de Problemas (8)

- $f(n) = O(n!)$:
  - Um algoritmo de complexidade $O(n!)$ é dito de complexidade exponencial, apesar de $O(n!)$ ter comportamento muito pior do que $O(2^n)$.
  - Geralmente ocorrem quando se usa força bruta para a solução do problema.
  - $n = 20 \rightarrow 20! = 2432902008176640000$, um número com 19 dígitos.
  - $n = 40$ um número com 48 dígitos.

Ordem de complexidade: mais encontradas

Notação	Nome	Característica	Exemplo
$O(1)$	constante	independe do tamanho n da entrada	determinar se um número é par ou ímpar; usar uma tabela de dispersão (hash) de tamanho fixo
$O(\log n)$	logarítmica	o problema é dividido em problemas menores	**busca binária**
$O(n)$	linear	realiza uma operação para cada elemento de entrada	**busca sequencial**; soma de elementos de um vetor
$O(n \log n)$	log-linear	o problema é dividido em problemas menores e depois junta as soluções	heapsort, quicksort, **merge sort**
$O(n^2)$	quadrática	itens processados aos pares (geralmente loop aninhado)	bubble sort (pior caso); quick sort (pior caso); **selection sort**; insertion sort
$O(n^3)$	cúbica		multiplicação de matrizes n x n; todas as triplas de n elementos
$O(n^c)$, $c>1$	polinomial		caixeiro viajante por programação dinâmica
$O(c^n)$	exponencial	força bruta	todos subconjuntos de n elementos
$O(n!)$	fatorial	força bruta: testa todas as permutações possíveis	caixeiro viajante por força bruta

Em geral: $O(1) < O(\log n) < O(n) < O(n\log n) < O(n2) < O(n3) < O(c^n) < O(n!)$

Comparação de Funções de Complexidade

Função de custo	Tamanho n					
	10	20	30	40	50	60
n	0,00001 s	0,00002 s	0,00003 s	0,00004 s	0,00005 s	0,00006 s
n^2	0,0001 s	0,0004 s	0,0009 s	0,0016 s	0,0.35 s	0,0036 s
n^3	0,001 s	0,008 s	0,027 s	0,64 s	0,125 s	0.316 s
n^5	0,1 s	3,2 s	24,3 s	1,7 min	5,2 min	13 min
2^n	0,001 s	1 s	17,9 min	12,7 dias	35,7 anos	366 séc.
3^n	0,059 s	58 min	6,5 anos	3855 séc.	10^8 séc.	10^{13} séc.

Algoritmos Polinomiais

- **Algoritmo exponencial** no tempo de execução tem função de complexidade $O(c^n)$, $c > 1$.
- **Algoritmo polinomial** no tempo de execução tem função de complexidade $O(p(n))$, onde $p(n)$ é um polinômio
- A distinção entre estes dois tipos de algoritmos torna-se significativa quando o tamanho do problema a ser resolvido cresce.
- Por isso, os algoritmos polinomiais são muito mais úteis na prática do que os exponenciais
- Algoritmos exponenciais são geralmente simples variações de pesquisa exaustiva.
- Algoritmos polinomiais são geralmente obtidos mediante entendimento mais profundo da estrutura do problema

Algoritmos Polinomiais × Algoritmos Exponenciais

- Um problema é considerado:
  - intratável: se não existe um algoritmo polinomial para resolvê-lo.
  - bem resolvido: quando existe um algoritmo polinomial para resolvê-lo.
- A distinção entre algoritmos polinomiais eficientes e algoritmos exponenciais ineficientes possui várias exceções.
- Exemplo: um algoritmo com função de complexidade $f(n) = 2^n$ é mais rápido que um algoritmo $g(n) = n^5$ para valores de n menores ou iguais a 20.

Algoritmos Polinomiais × Algoritmos Exponenciais

- Também existem algoritmos exponenciais que são muito úteis na prática

- Exemplo: o algoritmo Simplex para programação linear possui complexidade de tempo exponencial para o pior caso mas executa muito rápido na prática.

- Tais exemplos não ocorrem com frequência na prática, e muitos algoritmos exponenciais conhecidos não são muito úteis.

Exemplo de Algoritmo Exponencial

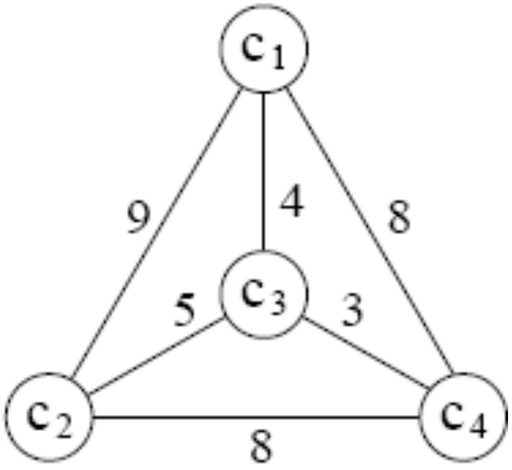
- Um **caixeiro viajante deseja visitar n cidades** de tal forma que sua viagem inicie e termine em uma mesma cidade, e cada cidade deve ser visitada uma única vez.

- Supondo que sempre há uma estrada entre duas cidades quaisquer, o problema é encontrar a menor rota para a viagem.

- A figura ilustra o exemplo para quatro cidades c1, c2, c3, c4, em que os números nas arestas indicam a distância entre duas cidades

Exemplo de Algoritmo Exponencial

- O percurso <c1, c3, c4, c2, c1> é uma solução para o problema, cujo percurso total tem distância 24.



Exemplo de Algoritmo Exponencial

- Um algoritmo simples seria verificar todas as rotas e escolher a menor delas.
- Há (n – 1)! rotas possíveis e a distância total percorrida em cada rota envolve n adições, logo o número total de adições é n!.
- No exemplo anterior teríamos 24 adições.
- Suponha agora 50 cidades: o número de adições seria $50! \approx 10^{64}$.
- Em um computador que executa 10^9 adições por segundo, o tempo total para resolver o problema com 50 cidades seria maior do que 10^{45} séculos só para executar as adições.
- O problema do caixeiro viajante aparece com frequência em problemas relacionados com transporte, mas também aplicações importantes relacionadas com otimização de caminho percorrido

Técnicas de Análise de Algoritmos

- Determinar o tempo de execução de um programa pode ser um problema matemático complexo;
- Determinar a ordem do tempo de execução, sem preocupação com o valor da constante envolvida, pode ser uma tarefa mais simples.
- A análise utiliza técnicas de matemática discreta, envolvendo contagem ou enumeração dos elementos de um conjunto:
  - manipulação de somas,
  - produtos,
  - permutações,
  - fatoriais,
  - coeficientes binomiais,
  - solução de equações de recorrência.

Análise do Tempo de Execução

- Comando de atribuição, de leitura ou de escrita: O(1).
- Sequencia de comandos: determinado pelo maior tempo de execução de qualquer comando da sequencia
- Comando de decisão: tempo dos comandos dentro do comando condicional, mais tempo para avaliar a condição, que é O(1).
- Anel: soma do tempo de execução do corpo do anel mais o tempo de avaliar a condição para terminação (geralmente O(1)), multiplicado pelo número de iterações.

Análise do Tempo de Execução (2)

- Procedimentos não recursivos:
  - Comece pelas funções que não chamam nenhuma outra função
  - Depois analise funções que chamam apenas funções analisadas no passo anterior
  - E assim sucessivamente até chegar ao programa principal (main)

Procedimento não Recursivo

Algoritmo Seleção:

- Seleciona o menor elemento do conjunto.
- Troca este com o primeiro elemento v[0].
- Repita as duas operações acima com os
  - n − 1 elementos restantes, depois com os
  - n − 2, até que reste apenas um.

```
package cap1;
public class Ordenacao {
  public static void ordena (int v[], int n) {
(1) for (int i = 0; i < n – 1; i++) {
(2)     int min = i;
(3)     for (int j = i + 1; j < n; j++)
(4)         if (v[j] < v[min])
(5)             min = j;
        /* Troca v[min] e v[i] */
(6)     int x = v[min];
(7)     v[min] = v[i];
(8)     v[i] = x;
    }
  }
}
```

Análise do Procedimento não Recursivo

- Anel Interno
  - Contém um comando de decisão, com um comando apenas de atribuição. Ambos levam tempo constante para serem executados.
  - Quanto ao corpo do comando de decisão, devemos considerar o pior caso, assumindo que ser sempre executado.
  - O tempo para incrementar o índice do anel e avaliar sua condição de terminação é $O(1)$.
  - O tempo combinado para executar uma vez o anel é $O(\max(1, 1, 1)) = O(1)$, conforme regra da soma para a notação O.
  - Como o número de iterações é $n - i$, o tempo gasto no anel é $O((n - i) \times 1) = O(n - i)$, conforme regra do produto para a notação O.

Análise do Procedimento não Recursivo

- Anel Externo
  - Contém, além do anel interno, quatro comandos de atribuição. O(max(1, (n – i), 1, 1, 1)) = O(n – i)
  - A linha (1) é executada n – 1 vezes, e o tempo total para executar o programa está limitado ao produto de uma constante pelo **somatório de (n – i) (soma de PA):**

$$\sum_{1}^{n-1}(n-i) = \frac{n(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2} = O(n^2)$$

$$T(n) = \frac{(a_1 + a_n).n}{2}$$

$$T(n) = \frac{(1 + n - 1).(n - 1)}{2}$$

$$T(n) = \frac{n.(n - 1)}{2}$$

$$T(n) = \frac{n^2 - n}{2}$$

$$T(n) = \frac{n^2}{2} - \frac{n}{2}.$$

$$T(n) = 1 + 2 + 3 + \ldots + n - 1.$$

Questão Poscomp 2019

QUESTÃO 25 – Considere a seguinte função em C:

```
void funcao(int n){
   int i,j;
   for (i=1; i<=n; i++)
      for(j=1; j<log(i); j++)
         printf("%d",i+j)
}
```

A complexidade dessa função é:

A) $\Theta(n)$
B) $\Theta(n \log n)$
C) $\Theta(\log n)$
D) $\Theta(n^2)$
E) $\Theta(n^2 \log n)$

Questão Poscomp 2019

QUESTÃO 22 – Considere as seguintes funções:

$$f(n) = 2^n$$

$$g(n) = n!$$

$$h(n) = n^{logn}$$

Assinale a alternativa correta a respeito do comportamento assintótico de f(n), g(n) e h(n).

A) $f(n) = O\big(g(n)\big); g(n) = O\big(h(n)\big).$
B) $f(n) = \Omega\big(g(n)\big); g(n) = O\big(h(n)\big).$
C) $g(n) = O\big(f(n)\big); h(n) = O\big(f(n)\big).$
D) $h(n) = O\big(f(n)\big); g(n) = \Omega\big(f(n)\big).$
E) Nenhuma das anteriores.

Questão ENADE 2021

QUESTÃO 20

Observe o código abaixo escrito na linguagem C.

```c
1   #include <stdio.h>
2   #define TAM 10
3   int funcao1(int vetor[], int v){
4       int i;
5       for (i = 0; i < TAM; i++){
6           if (vetor[i] == v)
7               return i;
8       }
9       return -1;
10  }
11  int funcao2(int vetor[], int v, int i, int f){
12      int m = (i + f) / 2;
13      if (v == vetor[m])
14          return m;
15      if (i >= f)
16          return -1;
17      if (v > vetor[m])
18          return funcao2(vetor, v, m+1, f);
19      else
20          return funcao2(vetor, v, i, m-1);
21  }
22  int main(){
23      int vetor[TAM] = {1, 3, 5, 7, 9, 11, 13, 15, 17, 19};
24      printf("%d - %d", funcao1(vetor, 15), funcao2(vetor, 15, 0, TAM-1));
25      return 0;
26  }
```

A respeito das funções implementadas, avalie as afirmações a seguir.

I. O resultado da impressão na linha 24 é: 7 - 7.

II. A função funcao1, no pior caso, é uma estratégia mais rápida do que a funcao2.

III. A função funcao2 implementa uma estratégia iterativa na concepção do algoritmo.

É correto o que se afirma em

A) I, apenas.
B) III, apenas.
C) I e II, apenas.
D) II e III, apenas.
E) I, II e III.

[POSCOMP] 2023

QUESTÃO 21 – Sobre os conceitos de complexidade de algoritmos, é correto afirmar que:

A) O espaço requerido por um algoritmo sobre uma dada entrada pode ser medido pelo número de execuções de algumas operações.
B) A complexidade de tempo usa como medida de desempenho a quantidade de memória necessária para a execução do algoritmo.
C) A complexidade média é definida pelo crescimento da complexidade para entradas suficientemente grandes.
D) A complexidade assintótica dá o valor esperado: a média dos esforços, levando em conta a probabilidade de ocorrência de cada entrada.
E) A complexidade pessimista de um algoritmo fornece seu desempenho no pior caso: o pior desempenho que se pode esperar. Aqui, pode-se considerar os desempenhos sobre todas as entradas com tamanho n.

Referências

- Thomas Cormen, Charles Leiserson and Ronald Rivest. Introduction to Algorithms. Second Edition. McGraw-Hill, 2007.
- Udi Mamber. Introduction to Algorithms: A Creative Approach. Addison-Wesley, 1989.
- Nivio Zizanni. Projeto de Algoritmos com implementações em Java e C++. Tomson, 2006.
- ASCENCIO, Ana Fernanda Gomes; ARAUJO, Graziela Santos. Estruturas de Dados: algoritmos, análise da complexidade e implementações em JAVA e C/C++. São Paulo: Perarson Prentice Halt, v. 3, 2010.
- Notas de Aula Disciplina Estruturas de Dados II: Prof. Dr. Geraldo Braz. DEINF/UFMA
- Notas de Aula Disciplina Algoritmos e Estruturas de Dados II: Prof. Dr. Ítalo Cunha. Departamento de Ciência da Computação. UFMG.
- Notas de Aula Disciplina Análise e Projeto de Algoritmos. Prof. Tiago A. E. Ferreira. Departamento de Estatística e Informática da Universidade Federal Rural de Pernambuco