PCC104 - Projeto e Análise de Algoritmos

Marco Antonio M. Carvalho

Departamento de Computação
Instituto de Ciências Exatas e Biológicas
Universidade Federal de Ouro Preto

Conteúdo

Fonte

Este material é baseado nos livros

- T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009.
- S. Halim. *Competitive Programming*. 3rd Edition, 2013.
- Ian Parberry and William Gasarch. *Problems on Algorithms*. Second Edition, 2002.
- Ian Parberry *Lecture Notes on Algorithm Analysis and Complexity Theory*. Fourth Edition, 2001.

Licença

Cálculo do Tempo de Execução

Ian Parberry e William Gasarch, *Problems on Algorithms*

"*Algorithm analysis usually means 'give a big-O figure for the running time of an algorithm.' [...] This can be done by getting a big-O figure for parts of the algorithm and then combining these figures using the sum and product rules for big-O.*

Another useful technique is to pick an elementary operation, such as additions, multiplications, or comparisons, and observe that the running time of the algorithm is big-O of the number of elementary operations. [...] Then you can analyze the exact number of elementary operations as a function of n in the worst case."

Cálculo do Tempo de Execução

Importância

Os algoritmos permeiam toda a ciência da computação (entre outras ciências), independente da área de concentração.

O projeto de algoritmos é fortemente influenciado pela estimativa de seu comportamento.

Estamos interessados em algoritmos eficientes, ou pelo menos, "bem comportados".

Projeto

Antes de projetarmos um algoritmo, analisa-se o problema: suas características, sua complexidade e o contexto em que se encontra – o que determinará a exigência sobre tempo de execução e qualidade das soluções.

Depois desta análise, as decisões se concentram em qual tipo de algoritmo será utilizado, quais estruturas de dados e outros detalhes de implementação.

Cálculo do Tempo de Execução

Implementação

Uma vez tomadas todas as decisões anteriores, implementa-se o algoritmo.

Qual é o custo de se utilizar uma determinada implementação específica?

Devemos levar em consideração a memória necessária para armazenar as estruturas de dados e os trechos do código – quantas vezes cada um será executado?

Analisando Algoritmos

Como visto anteriormente, podemos analisar o comportamento assintótico de um algoritmo (ou implementação) de acordo com o tamanho da entrada.

Se aplicarmos a mesma métrica a diferentes algoritmos para um mesmo problema, podemos compará-los de uma maneira adequada.

Analisando Algoritmos pt. 2

É necessário termos em mente de que se trata de uma análise teórica.

Na prática, outros fatores podem influenciar o desempenho de uma implementação de um algoritmo:

- Otimizações realizadas pelo compilador;
- Características do sistema operacional;
- Características de *hardware*.

Além disto, algumas simplificações são feitas nesta análise teórica, como veremos a seguir.

Comparando Algoritmos

Recomenda-se comparar algoritmos com complexidade dentro de uma mesma ordem de grandeza por meio de experimentos computacionais.

Desta forma, os custos reais e outros não aparentes se tornam claros.

Cálculo do Tempo de Execução

Como Medir?

Consideramos um conjunto de instruções com custos especificados, normalmente, só as instruções mais significativas.

Definimos uma função de custo ou função de complexidade T.

T(n) é a medida de custo da execução de um algoritmo para uma instância de tamanho n:

- A função de complexidade de tempo T(n) mede o tempo[a] necessário para executar um algoritmo.
- A função de complexidade de espaço T(n) mede a quantidade de memória necessária para executar um algoritmo.

[a] Não o tempo medido no relógio, mas quantas vezes operações relevantes serão executadas.

Cálculo do Tempo de Execução

Limitante Inferior

Dado um determinado problema P, chamamos de limite inferior (ou *lower bound*) LB(P) a complexidade mínima necessária para resolvê-lo.

Limitante Superior

Dado um determinado problema P, chamamos de limite superior (ou *upper bound*) UB(P) a complexidade do melhor algoritmo conhecido que o resolve.

Limitantes Superior e Inferior

Um determinado problema é considerado **computacionalmente resolvido** se UB(P) $\in \Theta$ (LB(P)).

Cálculo do Tempo de Execução

Exemplo

Consideremos o seguinte problema computacional: Dado um vetor A de n números inteiros, determine o maior valor entre eles.

Tópicos da Análise

1. Quais operações são relevantes?
2. Quais são os limitantes superior e inferior para este problema?
3. Quais são as características das perspectivas da análise?

Cálculo do Tempo de Execução

```
1 int arrayMax(int A[ ], int n)
2 {
3     int currentMax = A[0];
4     for(int i=1; i<n; i++){
5         if(A[i] > currentMax)
6             currentMax = A[i];
7     }
8     return currentMax;
9 }
```

Como calcular, em função de n, o número máximo de operações realizadas por este algoritmo, considerando as diferentes perspectivas?

Custos

A contribuição de cada instrução para o tempo de execução é o produto de seu custo individual e o número de vezes que é executada:

- Uma operação com custo c_1 executada uma vez contribui com c_1.
- Uma operação com custo c_2 executada n vezes contribui com $c_2 n$.
- Um laço (*for*, *while*) que termina de maneira usual contribui com o produto de uma constante e a quantidade de vezes que foi executado
  - Operações de atribuição, incremento e comparação são contados como uma única constante.
  - A comparação do laço é sempre executada uma vez mais, para determinar o seu fim
    - Por exemplo, um laço de 2 até n é executado n vezes.

Custos pt 2

- Um desvio condicional (*if*, *switch*) fora do cabeçalho de um laço é contado como tempo constante.
- Uma chamada para uma função tem complexidade correspondente à complexidade da execução da função.
- Uma função recursiva tem sua complexidade definida em termos da recorrência associada.

Instruções contidas dentro de laços são executadas repetidas vezes, o que deve ser levado em consideração.

O tempo de execução é, portanto, a soma destes produtos referentes a cada instrução do algoritmo.

Cálculo do Tempo de Execução

```
1 int arrayMax(int A[ ], int n)
2 {
3     int currentMax = A[0];// 1
4     for(int i=1; i<n; i++){// 1 + (n - 1) [a] + 1
5         if(A[i] > currentMax)// 1
6             currentMax = A[i];// 1
7     }
8     return currentMax;// 1
9 }
```

[a] 1 atribuição inicial e repete n-1 vezes (1 comparação, 1 incremento), no máximo. Adicionalmente, 1 última comparação

Análise Assintótica

No pior caso, são realizadas **4+4×(n-1)** operações (para $n \geq 1$), logo, o algoritmo é $O(n)$.

Cálculo do Tempo de Execução

Considerações

O algoritmo **arrayMax** executa $4n$ operações primitivas, excluindo os termos de mais baixa ordem.

Sejam a e b os tempos de execução das instruções mais rápida e mais lenta da arquitetura utilizada, respectivamente.

Seja T(n) o tempo real de execução do pior caso de **arrayMax**.

Temos que $a \times 4n \leq T(n) \leq b \times 4n$, portanto, T($n$) é delimitada por duas funções lineares.

A linearidade de T(n) é uma propriedade intrínseca de **arrayMax**. Por exemplo, o ambiente de *hardware* ou *software* apenas alterariam T(n) por uma constante, porém, a linearidade se manteria.

Propriedade

Cada algoritmo possui uma taxa de crescimento que lhe é intrínseca.

Otimalidade de um Algoritmo

Teorema - Limitante Inferior para Encontrar o Maior Elemento

Qualquer algoritmo para encontrar o maior elemento de um conjunto com n elementos ($n \geq 1$), faz pelo menos n-1 comparações.

Prova

Cada um dos n-1 elementos deve ser mostrado, por meio de comparações, ser menor do que algum outro elemento, logo, n-1 comparações são necessárias.

Otimalidade

Se o limitante inferior para encontrar o menor elemento é igual ao limitante superior, temos que o problema é computacionalmente resolvido e o algoritmo é **ótimo**.

Otimalidade de um Algoritmo

Otimalidade e Alternativas

Embora **arrayMax** seja um algoritmo ótimo, é possível que haja algum algoritmo de melhor performance para o mesmo problema?

Recapitulando... Perspectivas

Definição

Além do ambiente computacional, o comportamento de um algoritmo pode variar de acordo com o comportamento da entrada (tamanho, estrutura, etc.), o que gera diferentes perspectivas.

Melhor Caso

A entrada está organizada de maneira que o algoritmo levará o tempo mínimo para resolver o problema.

Pior Caso

A entrada está organizada de maneira que o algoritmo levará o tempo máximo para resolver o problema.

Caso Médio

A entrada está organizada de maneira que o algoritmo levará um tempo médio para resolver o problema.

Cálculo do Tempo de Execução

Perspectivas

Como visto, o pior caso de **arrayMax** é linear, ou seja, $O(n)$.

Qual é a complexidade de seu melhor caso?

E a complexidade de seu caso médio?

Outro Exemplo

Consideremos o problema de encontrar o maior **e** o menor elemento de um vetor de inteiros A, de tamanho n, com $n \geq 1$.

Vejamos um algoritmo de exemplo, em que T(n) se baseia no número de **comparações** entre os elementos de A.

Cálculo do Tempo de Execução

```
1 int maxMin1(int A[ ], int n, int max, int min)
2 {
3     max = A[0];
4     min = A[0];
5     for(int i=1; i<n; i++){// n - 1
6         if(A[i] > max)
7             max = A[i];
8         if(A[i] < min)
9             min = A[i];
10    }
11 }
```

Análise

Temos que o número de comparações realizadas é T(n) = 2(n-1) para n>0, para o melhor caso, pior caso e caso médio.
No entanto, este algoritmo pode ser melhorado.

Cálculo do Tempo de Execução

```
1  int maxMin2(int A[ ], int n, int max, int min)
2  {
3      max = A[0];
4      min = A[0];
5      for(int i=1; i<n; i++){// n - 1
6          if(A[i] > max)
7              max = A[i];
8          else if(A[i] < min)
9              min = A[i];
10     }
11 }
```

Observação

Para esta nova versão do algoritmo, as perspectivas mudaram?

Cálculo do Tempo de Execução

Melhor Caso

Os elementos estão em ordem crescente, logo, o número de comparações é $T(n) = n\text{-}1$.

Pior Caso

Os elementos estão em ordem decrescente, logo, o número de comparações é $T(n) = 2(n\text{-}1)$.

Caso Médio

Supõe-se uma distribuição de probabilidades sobre o conjunto de entradas de tamanho n.

É comum supor uma distribuição em que quaisquer entradas são igualmente prováveis, embora isso não seja sempre verdade.

Esta análise geralmente é mais elaborada do que as duas anteriores.

Caso Médio pt. 2

Podemos considerar uma distribuição dos elementos de A de maneira que A[i] será maior do que a variável **max** na metade dos casos. Ou seja, o primeiro *if* será executado $n-1$ vezes, e o *else* $\frac{n-1}{2}$ vezes.

Portanto, o número de comparações é $\mathsf{T}(n) = n - 1 + \frac{n-1}{2} = \frac{3n}{2} - \frac{3}{2}$, para $n>0$.

Observação

Este não é um algoritmo ótimo, embora seja melhor do que o primeiro.

Vejamos uma terceira versão de algoritmo.

Cálculo do Tempo de Execução

maxMin3 - Princípio

- Compare os elementos de A aos pares, separando-os em dois subconjuntos A^-, o conjunto dos menores elementos e A^+, o conjunto dos maiores elementos.
- Obtenha o maior elemento comparando os $\lceil \frac{n}{2} \rceil$ -1 elementos do conjunto A^+.
- Obtenha o menor elemento comparando os $\lceil \frac{n}{2} \rceil$ -1 elementos do conjunto A^-.

Qual é a complexidade de cada perspectiva deste algoritmo?

Cálculo do Tempo de Execução

```
1  int maxMin3(int A[ ], int n, int max, int min)
2  {
3      if(n%2 != 0){
4          A[n+1] = A[n];
5          n = n+1;
6      }
7      max = A[0];
8      min = A[1];
9      if(A[0] < A[1]){
10         max = A[1];
11         min = A[0];
12     }
```

Cálculo do Tempo de Execução

```
1     for(int i=2; i<n-1; i+=2){
2         if(A[i] > A[i+1]){
3             if(A[i] > max)
4                 max = A[i];
5             if(A[i+1] < min)
6                 min = A[i+1];
7         }else{
8             if(A[i] < min)
9                 min = A[i];
10            if(A[i+1] > max)
11                max = A[i+1];
12        }
13    }
14 }
```

Cálculo do Tempo de Execução

Observações

Os elementos de A são comparados dois a dois:

- Os elementos maiores são comparados com **max**;
- Os elementos menores são comparados com **min**.

Quando n é ímpar, o último elemento de A é duplicado, por simplicidade.

Complexidade Algoritmo

O número de comparações é $\mathsf{T}(n) = \frac{n}{2} + \frac{n-2}{2} + \frac{n-2}{2} = \frac{3n}{2} - 2$ para $n>0$ e para todas as perspectivas.

Complexidade Implementação

O número de comparações é $\mathsf{T}(n) = 1 + \frac{n-2}{2} + \frac{n-2}{2} + \frac{n-2}{2} = \frac{3n}{2} - 2$ para $n>0$ e para todas as perspectivas.

Cálculo do Tempo de Execução

O Bom, o Mau e o Feio

Comparemos as três versões de algoritmos apresentadas

- De uma maneira geral, **maxMin2** e **maxMin3** são superiores a **maxMin1**.
- **maxMin2** é superior a **maxMin3** no melhor caso.
- **maxMin3** é superior a **maxMin2** com relação ao pior caso.
- **maxMin2** e **maxMin3** são bastante próximos quando ao caso médio.

Qual algoritmo você escolheria?

	Melhor Caso	Caso Médio	Pior Caso
maxMin1	$2(n\text{-}1)$	$2(n\text{-}1)$	$2(n\text{-}1)$
maxMin2	$n\text{-}1$	$\frac{3n}{2} - \frac{3}{2}$	$2(n\text{-}1)$
maxMin3	$\frac{3n}{2} - 2$	$\frac{3n}{2} - 2$	$\frac{3n}{2} - 2$

Tabela: Comparação entre algoritmos.

Cálculo do Tempo de Execução

Análise Assintótica e Perspectivas

A notação Ω é utilizada para expressar o **melhor** caso de um algoritmo, ou seja, para definir o limite inferior para o tempo de execução deste algoritmo.

A notação O é utilizada para expressar o **pior** caso de um algoritmo, ou seja, para definir o limite superior para o tempo de execução deste algoritmo.

Exercício

Vimos a análise da quantidade de comparações de cada algoritmo em cada perspectiva. Faça a análise assintótica de cada um dos três algoritmos anteriores.

Com base apenas na análise assintótica, qual algoritmo você escolheria? O que mudou em relação à análise anterior?

Comparando Algoritmos

Comparação Justa?

Podemos comparar algoritmos utilizando as funções de complexidade de espaço e tempo, negligenciando as constantes de proporcionalidade.

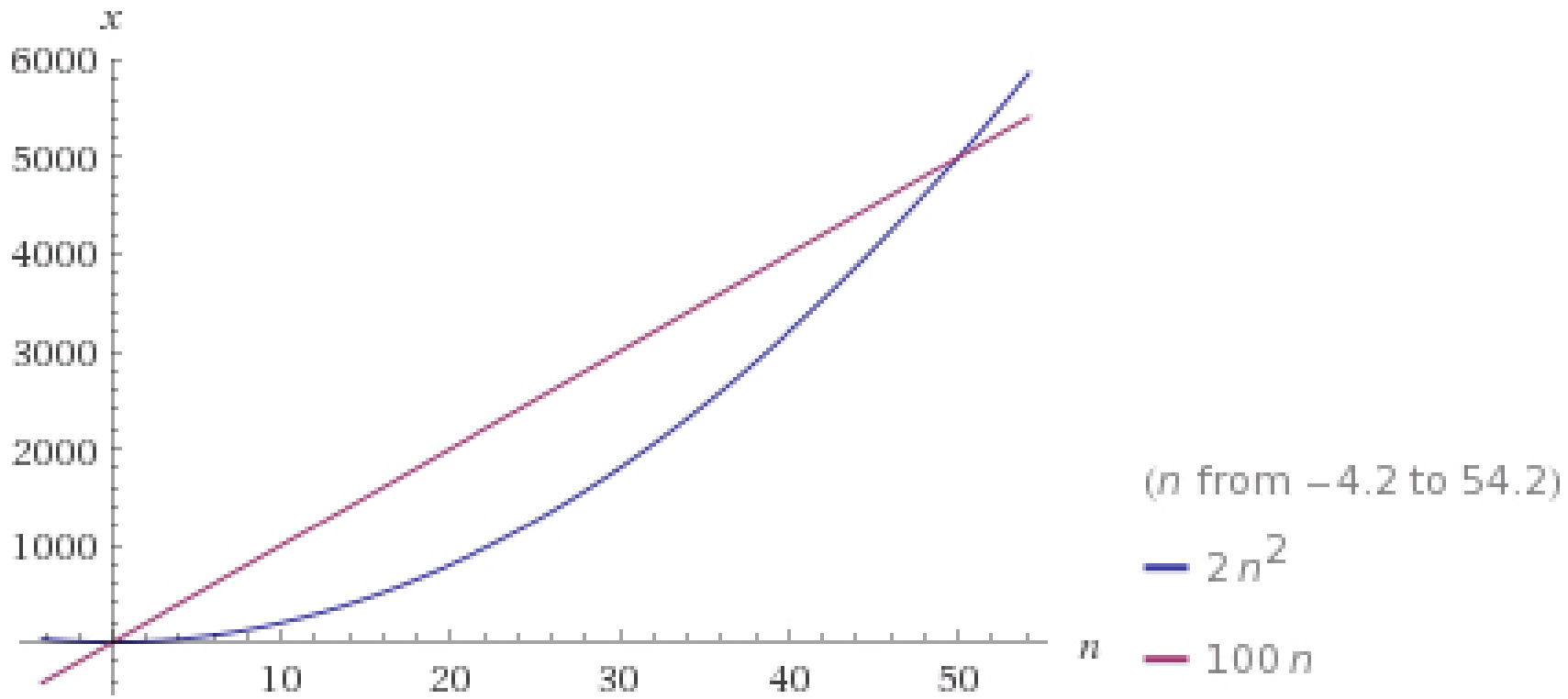
Desta forma, um algoritmo $O(n^2)$ é pior que outro $O(n)$, ambos para o mesmo problema. Contudo, as constantes de proporcionalidade podem revelar fatos escondidos.

Exemplo

Suponha dois algoritmos: um exige $100n$ unidades de tempo e outro exige $2n^2$ unidades de tempo. Dependendo do tamanho do problema, o melhor algoritmo pode variar:

- Para $n < 50$, o segundo algoritmo é melhor que o primeiro;
- Se a quantidade de dados for pequena, é preferível optar pelo segundo;
- Entretanto, o tempo de execução do segundo algoritmo cresce mais rapidamente que o tempo de execução do primeiro.

Comparando Algoritmos



Classes de Comportamento Assintótico

T(n) = $O(1)$, ou Complexidade Constante

A complexidade do algoritmo independe do tamanho da entrada, ou seja n.

Ocorre quando as instruções do algoritmo são executadas um número constante de vezes.

Exemplo

Laços de repetição com condição de parada fixa.

Classes de Comportamento Assintótico

T(n) = $O(logn)$, ou Complexidade Logarítmica

Ocorre tipicamente em algoritmos que dividem o problema original em subproblemas e algoritmos que usam árvores.

O tempo de execução pode ser considerado menor do que uma constante grande. Supondo logaritmo na base 2:

- $n = 1.000$, $log_2\ n \approx 10$;
- $n = 1.000.000$, $log_2\ n \approx 20$.

Exemplo

Busca Binária.

Classes de Comportamento Assintótico

T(n) = $O(n)$, ou Complexidade Linear

Normalmente, os algoritmos de complexidade linear realizam alguma operação leve sobre cada elemento da entrada.

É a melhor situação possível para um algoritmo que precisa processar n elementos de entrada ou produzir n elementos de saída.

Exemplo

Pesquisa Sequencial.

Classes de Comportamento Assintótico

T(n) = $O(n\ log\ n)$, ou Complexidade Linear Logarítmica

Ocorre tipicamente em algoritmos que dividem o problema original em subproblemas, resolvem cada um deles e depois agrupam os resultados em um só.

Muito associado ao paradigma **Divisão e Conquista**.

Supondo logaritmo na base 2:

- n = 1.000.000, $n\ log_2\ n \approx$ 20.000.000;
- n = 2.000.000, $n\ log_2\ n \approx$ 42.000.000.

Exemplo

Merge Sort.

Classes de Comportamento Assintótico

T(n) = $O(n^2)$, ou Complexidade Quadrática

Ocorre quando os elementos da entrada são processados aos pares, como em laços aninhados.

Sempre que n dobra, a complexidade é multiplicada por 4.

Embora pareçam ruins, são úteis para resolver instâncias relativamente pequenas.

Exemplos

Insertion Sort, Bubble Sort e Quick Sort (pior caso).

Classes de Comportamento Assintótico

T(n) = $O(n^3)$, ou Complexidade Cúbica

Geralmente são úteis apenas para resolver problemas relativamente pequenos, ou quando comprovadamente o pior caso não é frequente.

Sempre que n dobra, a complexidade é multiplicada por 8.

Para alguns problemas, os melhores algoritmos conhecidos são cúbicos.

Exemplos

Multiplicação de Matrizes, Algoritmos para Problemas NP-Difíceis.

Classes de Comportamento Assintótico

T(n) = $O(k^n)$, ou Complexidade Exponencial

Não são úteis do ponto de vista prático caso esta complexidade seja uma perspectiva frequente.

São associados a **força bruta** (ou **busca exaustiva**).

Sempre que n dobra, a complexidade se eleva ao quadrado.

Exemplos

Simplex (pior caso – não frequente) e Algoritmos para o Problema do Caixeiro Viajante.

Classes de Comportamento Assintótico

T(n) = $O(n!)$, ou Complexidade Fatorial

Também é considerado como complexidade exponencial, embora apresente um comportamento muito pior do que $O(k^n)$.

Também associado ao paradigma de força bruta e à geração de permutações.

Cresce muito rapidamente:

- $n = 20$, $n!$ resulta em um número com 19 dígitos;
- $n = 40$, $n!$ resulta em um número com 48 dígitos.

Exemplo

Geração de permutações.

Dúvidas?