ESTRUTURAS DE DADOS

Complexidade II

Introdução

- Suponha duas soluções para um mesmo problema
  - Qual delas é a melhor?
- Suponha dois algoritmos que realizem a mesma tarefa
  - Qual deles é o melhor?
    - O mais rápido e que consome menos memória?
    - Em computação existem parâmetros que nos permitem comparar e medir quanto uma solução pode ser melhor do que outra

Introdução

- Limitações
  - Medir o tempo de execução de um algoritmo não é uma tarefa imediata já que vários fatores influenciam na medição
- Solução
  - Considerar uma arquitetura computacional padrão
    - um único processador e acesso randômico de memória (RAM)
  - Estimar o tempo de execução de um algoritmo
    - contar o número de vezes que cada linha do algoritmo é executada e o tempo de processamento de cada linha

Introdução

- Por exemplo
  - Suponha que uma máquina realiza somas, comparações e atribuições em c_1, c_2 e c_3 milissegundos, respectivamente
  - Calcule quantos milissegundos o código para comparar duas sequências de *n* caracteres (A e B) leva para ser executado

```
1.  int i = 0;
2.  while ((i < n) && (A[i] == B[i]))
3.        i++;
```

  - Resposta: $n*(c_1 + 2*c_2) + 2*c_2 + c_3$ milissegundos já que a linha 1 consome c_3 milissegundos e as linhas 2 e 3 consomem $2(n+1)c_2$ e nc_1 milissegundos, respectivamente

Introdução

- Algumas considerações a respeito do exemplo
  - Se esse algoritmo fosse executado em outra máquina c_1, c_2 e c_3 seriam diferentes, mas o cálculo do tempo de processamento seria o mesmo
  - O tempo de processamento do algoritmo aumenta de acordo com *n*
  - No pior caso (quando as duas sequências são idênticas) o loop é executado *n* vezes, mas em outras situações o tempo será menor

Introdução

- Conclusão
  - Identificar a complexidade computacional de um algoritmo (o consumo de tempo e espaço requerido para sua execução) não é uma tarefa trivial pois depende de:
    - parâmetros de entrada
    - velocidade de processamento
    - muitas possibilidades a considerar que vão desde o pior até o melhor caso para o processamento

Notação Assintótica

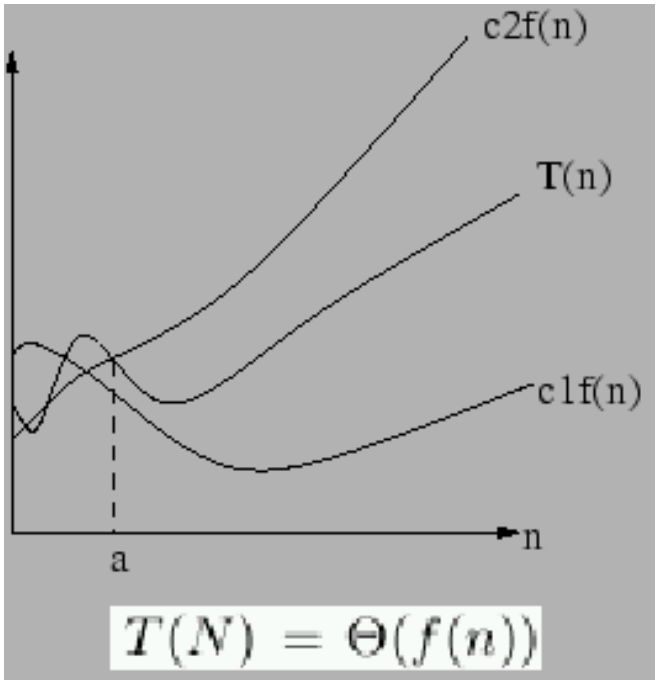
- Com o objetivo de equacionar as dificuldades apontadas acima utiliza-se o que é hoje conhecida como notação assintótica
- Além de desvincular a análise do algoritmo das propriedades da máquina, permite tratar as diferentes possibilidades de desempenho decorrentes do padrão de entrada apresentado
- Viabiliza uma análise desde o pior até o melhor desempenho de um algoritmo

Notação Θ

- Definição
  - Para uma dada função positiva f(n), denotamos por Θ(f(n)) o conjunto de funções definido como:
    - Θ(f(n)) = { T(n) | existem constantes positivas c_1, c_2 e tais que $0 \leq c_1 f(n) \leq T(n) \leq c_2 f(n)$, para todo $n \geq a$}
  - Ex: a função $1/2n^2 - 3n = \Theta(n^2)$
    - $c_1 n^2 \leq 1/2n^2 - 3n \leq c_2 n^2$ para $c_1 = 1/14$, $c_2 = 1/2$ e $a = 7$

Notação Θ

- A figura a seguir mostra a ideia intuitiva da definição

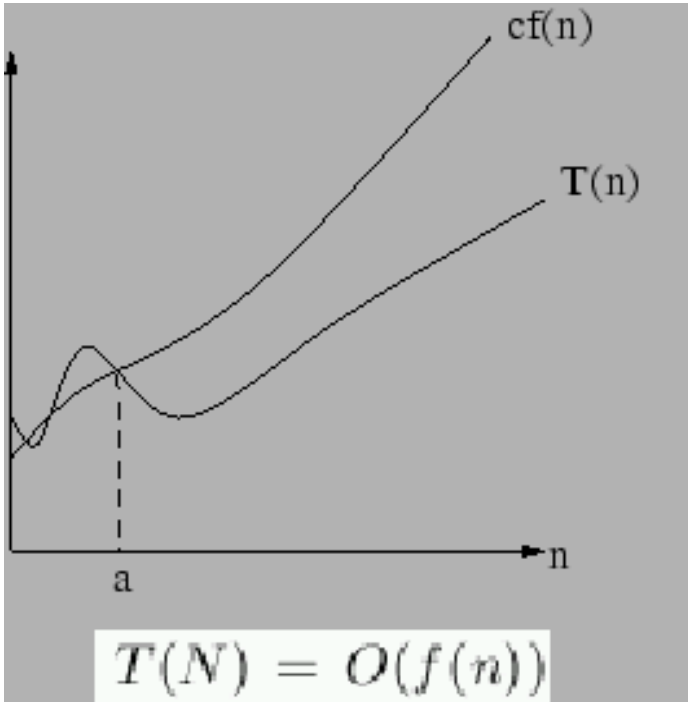


Notação O

- Utilizada sempre que pretendemos estimar o pior desempenho de um algoritmo
  - Definição
    - Para uma dada função positiva f(n), denotamos como O(f(n)) o seguinte conjunto de funções
    - O(f(n)) = { T(n) | existem constantes positivas c e a tais que $0 \leq T(n) \leq c^* f(n)$, para todo $n \geq a$}
- A notação O dá um limite superior assintótico para uma função

Notação O

- A figura a seguir mostra a ideia intuitiva que está por trás da definição acima, ou seja, para valores de *n* maiores ou iguais a *a*, o valor de T(*n*) é sempre menor ou igual a uma constante multiplicada por f(*n*)



Notação O

- Por exemplo, se $T(n)$ é o tempo computacional gasto por um algoritmo para realizar uma determinada tarefa no seu pior caso, em que n é o tamanho da entrada, dizer que $T(n) \in O(f(n))$ significa que, não importa qual seja o conjunto de dados de tamanho n escolhido, o tempo consumido pelo algoritmo será sempre menor do que uma constante vezes $f(n)$

Notação O

- Considere o código para cálculo da soma dos *n* primeiros números inteiros não negativos

```
1.   x = 0;
2.   i = 1;
3.   while (i <= n)
4.        x += i++;
```

- Sejam c_1 ,c_2 e c_3 os tempos computacionais na execução das linhas 1, 3 e 4 (a linha 2 consome o mesmo tempo que a linha 1)
- As linhas 3 e 4 serão executadas *n*+1 e *n* vezes, respectivamente

Notação O - Exemplo

- O tempo total gasto pelo código será
  - $T(n) = 2c_1 + (n+1)c_2 + nc_3 = n(c_2 + c_3) + 2c_1 + c_2$
- Se tomarmos
  - $c = \max(c_2 + c_3, 2c_1 + c_2)$, temos que
  - $T(n) \leq n(c+c/n) \leq n(2c)$, $n \geq 1$
- Portanto
  - $T(n) \in O(n)$ ou seja, para qualquer n, $T(n)$ é sempre menor que uma constante (no exemplo acima, $2c$) vezes n

Notação O - Exemplo

- Considerações
  - Note que as constantes c_1 ,c_2 e c_3 definidas pela velocidade de processamento da máquina ficam implicitamente embutidas na notação
  - O que a notação está nos dizendo é que o tempo de processamento do algoritmo irá aumentar linearmente com o valor de *n*, independentemente da máquina

Notação O - Cuidado

- Embora as constantes implicitamente representadas na notação assintótica permitam desprezar as particularidades da máquina, elas podem influenciar no desempenho de um algoritmo
- Suponha que dois algoritmos que desempenham a mesma tarefa para um conjunto de dados de tamanho *n* sejam
  - Algoritmo 1: $T_1(n) \in O(n)$
  - Algoritmo 2: $T_2(n) \in O(n^2)$
- Qual deles é o melhor?

Notação O - Cuidado

- Para valores grandes de n
  - O Algoritmo 1 é mais eficiente
- Mas se os valores de n são pequenos esse fato pode não ser verdadeiro. Por exemplo,
  - Se $T_1(n) = 10^3 n$ e $T_2(n) = n^2$
  - O Algoritmo 2 é mais rápido que o Algoritmo 1 para todo $n < 10^3$

Notação O

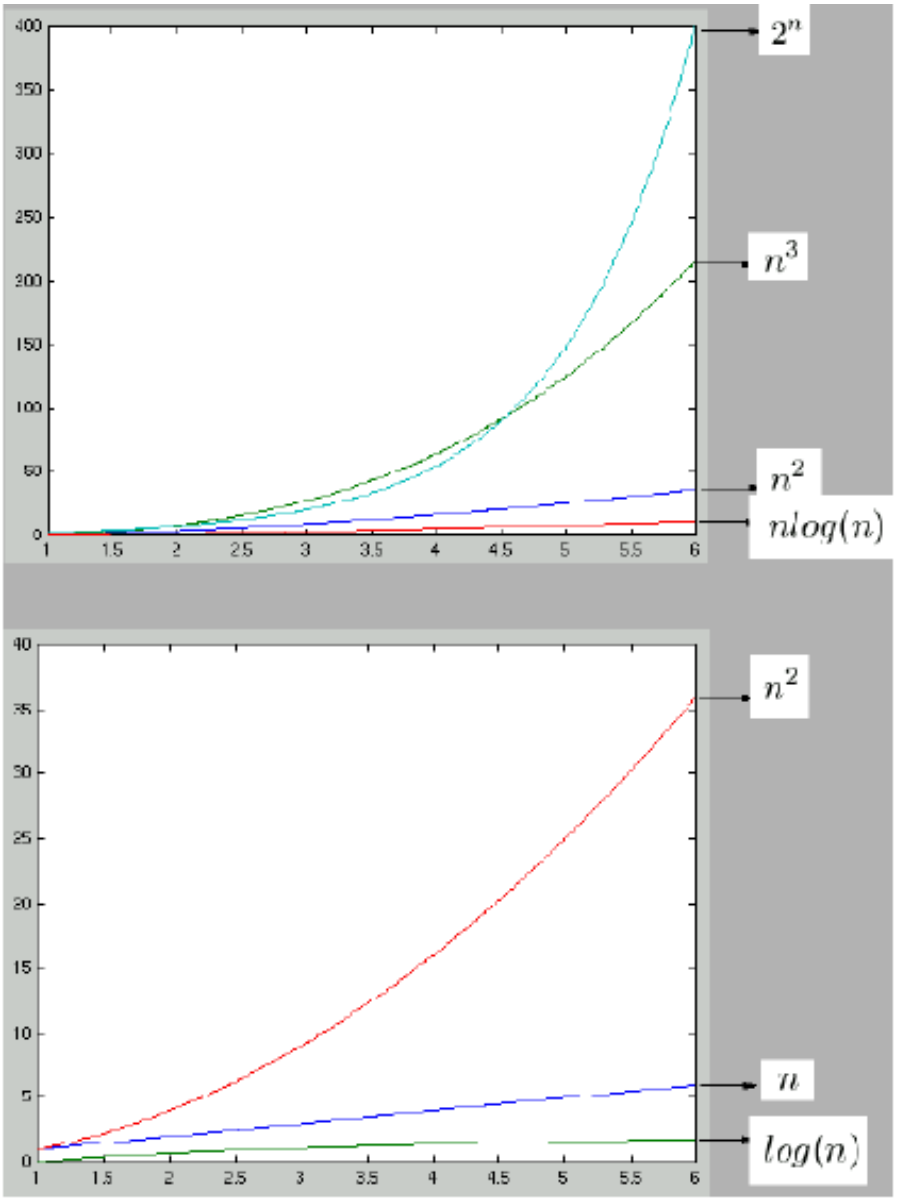
- Tempos computacionais mais comuns

$O(1) < O(\log(n)) < O(n) < O(n\log(n)) < O(n^2) < O(n^3) < O(2^n)$

  - $O(1)$: o número de operações executados é independente da entrada
  - $O(n^m)$: algoritmos polinomiais
  - $O(m^n)$ (como é o caso de $O(2^n)$): algoritmos exponenciais

Notação O

Como as ordens de magnitude das funções apresentadas anteriormente crescem com o valor de *n*

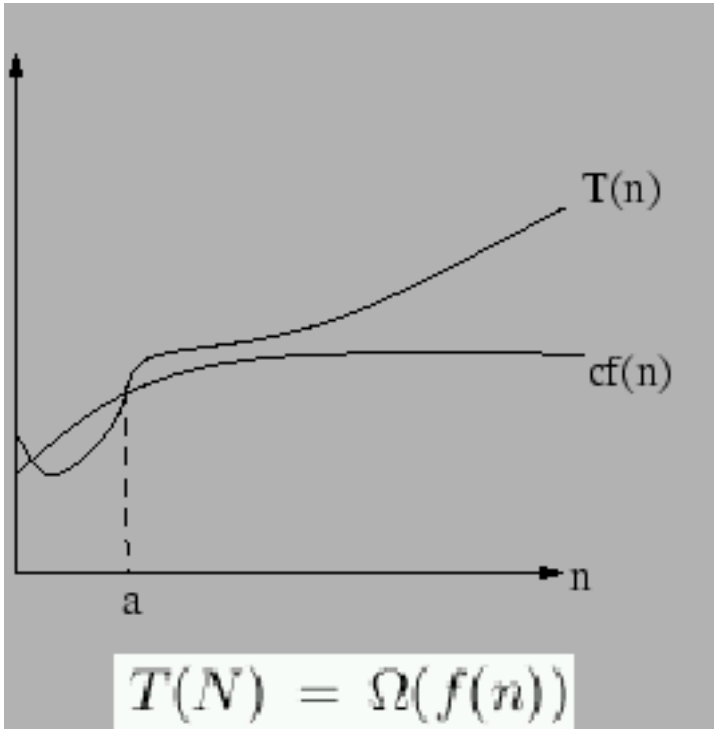


Notação Ω

- Da mesma forma que a notação O fornece um limite superior assintótico para uma função, a notação Ω (ômega) fornece um limite inferior assintótico
  - Definição
    - Para uma dada função positiva f(*n*), denotamos por Ω(f(*n*)) o conjunto de funções definido como:
    - Ω(f(*n*)) = { T(*n*) | existem constantes positivas *c* e *a* tais que $0 \leq c*f(n) \leq T(n)$, para todo $n \geq a$}
- Note que a notação Ω é utilizada para estudo do melhor caso

Notação Ω

- A figura a seguir mostra ideia intuitiva da definição acima, ou seja, para valores de *n* maiores ou iguais a *a*, o valor de T(*n*) é sempre maior ou igual a uma constante multiplicada por f(*n*)



Notação Ω - Exemplo

- Considere o código que procura um número inteiro *a* em um array *A* de tamanho *n*

```
1.    i = 0;
2.    while (i < n)
3.          if (A[i] != a)
4.              i++;
5.          else return(i);
6.    return(-1);
```

Notação Ω - Exemplo

- Sejam c_1 … c_6 os tempos computacionais para executar as linhas de 1 a 6 do algoritmo apresentado anteriormente
  - Melhor caso: o elemento procurado está na primeira posição do array. Nesse caso as linhas 1, 2, 3 e 5 serão executadas apenas uma vez e
  - $T(n) = c_1 + c_2 + c_3 + c_5$, ou seja, $T(n) = \Omega(1)$
  - Pior caso: o elemento procurado não está no array. Nesse caso as linhas 3 e 4 serão executadas n vezes e a linha 2, $n+1$ vezes, obtendo $T(n) = O(n)$

Análise do Caso Médio

- A análise de desempenho médio de um algoritmo é uma tarefa crucial em muitas situações, mas ...
  - O que é uma entrada média para um algoritmo?

Análise do Caso Médio

- Metodologia
  1. Encontrar uma distribuição de probabilidades para as entradas do problema,
  2. Calcular as complexidades para cada uma dessas entradas e
  3. Estimar o comportamento médio como a soma dos produtos de cada complexidade pela probabilidade de sua ocorrência
- Difícil de ser conduzida e até inviável em alguns casos
- Por isso, algumas análises de caso médio simplificam as hipóteses com relação aos dados de entrada visando facilitar a análise

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 em um número fixo de vezes.

Principais Classes de Problemas

- f(n) = O(log n)
  - Um algoritmo de complexidade O(log n) é dito ter 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.
  - A base do logaritmo muda pouco estes valores: quando n é 1 milhão, o $\log_2 n$ é 20 e o $\log_{10} n$ é 6.

Principais Classes de Problemas

- f(n) = O(n)
  - Um algoritmo de complexidade O(n) é dito ter 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 dobra.

Principais Classes de Problemas

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

Principais Classes de Problemas

- $f(n) = O(n^2)$
  - Um algoritmo de complexidade $O(n^2)$ é dito ter complexidade quadrática (polinomial).
  - 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

- $f(n) = O(n^3)$
  - Um algoritmo de complexidade $O(n^3)$ é dito ter complexidade cúbica (polinomial).
  - 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

- $f(n) = O(2^n)$
  - Um algoritmo de complexidade $O(2^n)$ é dito ter 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

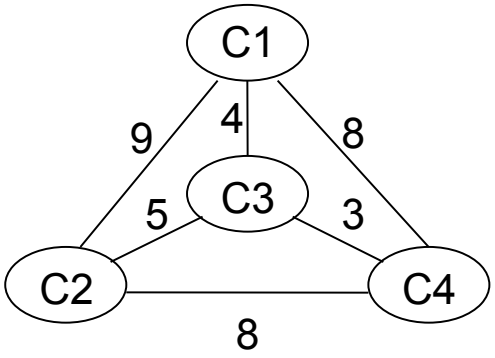
- f(n) = O(n!)
  - Um algoritmo de complexidade O(n!) é dito ter complexidade exponencial, apesar de O(n!) ter comportamento muito pior do que O(2^n).
  - Geralmente ocorrem quando se usa força bruta para a solução do problema.
  - $n = 20 \rightarrow 20! = 2432902008176640000$, um número com 19 dígitos.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

Polinomiais X Exponenciais

- Distinção significativa quando o tamanho do problema 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.
- Um problema é considerado:
  - intratável: se não existe um algoritmo polinomial para resolvê-lo.
  - bem resolvido: quando existe um algoritmo polinomial para resolvê-lo.
- Alg. Exponenciais podem ser úteis na prática
- Ex. Simplex para programação linear: possui complexidade de tempo exponencial para o pior caso mas executa muito rápido na prática.

Exemplo de Alg. 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.



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

Exemplo de Alg. 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 em aplicações importantes relacionadas com otimização de caminho percorrido.

Slides baseados nos livros

- Nivio Ziviani, **Projeto de Algoritmos com Implementações em PASCAL e C**, Cengage Learning, 3ª edição, 2011.
- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, **Algoritmos**, Campus, 3ª edição, 2012.