Complexidade de algoritmos
Notação Big-O

Prof. Tiago Massoni

Engenharia da Computação

Poli- UPE

Motivação

- O projeto de algoritmos é fortemente influenciado pelo estudo de seus comportamentos
- Problema analisado e decisões de projeto finalizadas
  - Então estudar opções de algoritmos, considerando os aspectos de tempo de execução e espaço ocupado
- Algoritmos encontrados em áreas como pesquisa operacional, otimização, teoria dos grafos, estatística, probabilidades, entre outras

Custo de algoritmos

- Determinando o menor custo possível para resolver problemas, temos a medida da dificuldade inerente para resolver o problema
- Quando o custo de um algoritmo é igual ao menor custo possível, o algoritmo é ótimo para a medida considerada
- Podem existir vários algoritmos 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

Medida de custo pela execução

- Normalmente inadequadas (resultados não são generalizados)
  - 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
- Apesar disso, argumentos a favor
  - Por exemplo, vários algoritmos distintos para resolver um mesmo tipo de problema, com custo de execução dentro de uma mesma ordem de grandeza

Medida do custo por modelo matemático

- Usa 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 das operações e considerar apenas as operações mais significativas (abstração)
- Ex.: algoritmos de ordenação.
  - Consideramos o número de comparações entre os elementos
  - Ignoramos as operações aritméticas, de atribuição e manipulações de índices, caso existam (além de operações de entrada e saída)

Modelo: função de complexidade

- Medição de custo de execução de um algoritmo
  - é comum definir uma função de custo ou função de complexidade f
  - f(n) é a medida do tempo necessário para executar um algoritmo para um problema de tamanho n
- Função de complexidade de tempo: f(n): tempo necessário para executar em um problema de tamanho n
- Função de complexidade de espaço: f(n): memória necessária para executar em um problema de tamanho n
- Utilizaremos f para denotar uma função de complexidade de tempo daqui para a frente
  - Não representa tempo diretamente, mas o número de vezes que determinada operação considerada relevante é executada.

Exemplo: maior elemento de vetor

```
public class Max {
  public static int max (int v[], int n){
    int max = v[0] ;
    for (int i = 1;i<n;i++)
      if (max < v[i]) max = v[i];
    return max;
  }
}
```

- Seja f uma função de complexidade tal que f(n) é o número de comparações entre os elementos de v, se v contiver n elementos.
- Logo $f(n) = n - 1$, para $n > 0$

Tamanho da entrada de dados (n)

- A medida do custo de execução de um algoritmo depende principalmente do tamanho da entrada dos dados (tamanho do problema)
- tempo de execução de um programa = 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
  - No caso do método max do programa do exemplo, o custo é uniforme sobre todos os problemas de tamanho n
  - Já para um algoritmo de ordenação, se os dados de entrada já estiverem quase ordenados, então o algoritmo pode ter que trabalhar menos

Casos

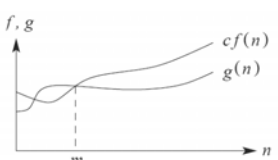
- Melhor caso: menor tempo de execução sobre todas as entradas de tamanho n
- Pior caso: maior tempo de execução sobre todas as entradas de tamanho n
  - Normalmente mais importante de analisar
  - Fácil de definir, define o limite da execução (garantia)
- Caso médio (ou caso esperado): média dos tempos de execução de todas as entradas de tamanho n
  - Supõe-se uma distribuição de probabilidades sobre entradas de tamanho n
  - Geralmente muito mais difícil de obter

Exemplo: achar registro em arquivo

- Cada registro contém uma chave única que é utilizada para recuperar registros do arquivo
  - Dada uma chave qualquer, localize o registro que contenha esta chave
- O algoritmo de pesquisa mais simples: pesquisa seqüencial
- Seja f uma função de complexidade tal que f(n) é o número de vezes que a chave de consulta é comparada com a chave de cada registro
  - melhor caso: $f(n) = 1$ (registro é o primeiro consultado)
  - pior caso: $f(n) = n$ (registro é o último consultado ou não está presente no arquivo)
  - caso médio: $f(n) = (n + 1)/2$

Notação Big-O (Ozão)

- Estuda-se o comportamento assintótico das **funções de custo** (para valores grandes de n)
- Ozão: modelo matemático para representar o pior caso de g(n)
- Escrevemos $g(n) = O(f(n))$ para expressar que f(n) domina assintoticamente g(n)
  - Lê-se g(n) é da ordem no máximo f(n)
- **Definição**: Uma função g(n) é O(f(n)) se existem duas constantes positivas c e m tais que $g(n) <= cf(n)$, para todo $n >= m$



Exemplos

- $g(n) = (n + 1)^2$
  - $g(n)$ é $O(n^2)$, quando m = 1 e c = 4
  - Isso porque $(n + 1)^2 <= 4n^2$ para $n >= 1$(m)
- $g(n) = 3n^3+2n^2+n$ é $O(n^3)$
  - $3n^3+2n^2+n <= 6n^3$, para $n>=0$
  - A função é também $O(n^4)$, entretanto esta afirmação é mais fraca do que dizer que g(n) é O(n3)

Exemplos

- $g(n) = \log_5 n$ é $O(\log n)$
  - O $\log_b n$ difere do $\log_c n$ por uma constante
  - Portanto a base de um logaritmo não importa no comportamento assintótico das funções

Classes de comportamento assintótico

- Se f é função de complexidade para algoritmo F, então O(f) é a complexidade assintótica do algoritmo F
- 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: $O(f(n)) = O(g(n))$

Exemplo de comparação

- Um programa leva 100n unidades de tempo para ser executado e outro leva $2n^2$
- Qual dos dois programas é melhor?
  - depende do tamanho do problema
- Para $n < 50$, o programa com tempo $2n^2$ é melhor do que o que possúi tempo 100n
  - Para problemas com entrada de dados pequena é preferível usar o programa cujo tempo de execução é $O(n^2)$
  - Entretanto, quando n cresce, o programa com tempo de execução $O(n^2)$ leva muito mais tempo que o programa $O(n)$

Classes de complexidade

- $f(n) = O(1)$ - complexidade constante
  - Uso do algoritmo independe de n
  - Instruções executadas um número fixo de vezes
- $f(n) = O(\log n)$ - complexidade logarítmica
  - Típico em algoritmos que transformam um problema em outros menores
  - Quando n é 1000, $\log_2 n=10$, quando n é 1 milhão, $\log_2 n=20$
- $f(n) = O(n)$ - complexidade linear
  - Em geral, um pequeno trabalho é realizado sobre cada elemento de entrada
  - Melhor situação possível para processar n elementos de entrada

Classes de complexidade

- $f(n) = O(n \log n)$
  - Algoritmos que quebram um problema em outros menores, resolvem cada um deles e juntam as soluções depois
  - Quando n é 1 milhão, $n\log_2 n$ é cerca de 20 milhões
- $f(n) = O(n^2)$ - complexidade quadrática.
  - Itens de dados processados aos pares, muitas vezes em um laço dentro de outro
  - Quando n é 1000, o número de operações é da ordem de 1 milhão
- $f(n) = O(n^3)$ - complexidade cúbica
  - Quando n é 100, o número de operações é da ordem de 1 milhão

Classes de complexidade

- $f(n) = O(2^n)$ - complexidade exponencial
  - **Força bruta** para resolver problemas
  - Quando n é 20, o tempo de execução é cerca de 1 milhão

Regras gerais de cálculo

- Laços
  - Custo das instruções x número de iterações
  - Se aninhados, começar o cálculo de dentro para fora
- if-else
  - Custo do teste + custo maior do (if/else)

Exemplos de algoritmos anteriores

- Listas com arrays
  - printList e find: $O(n)$
  - findKth: $O(1)$
  - insert e remove: $O(n)$
- Listas ligadas
  - printList e find: $O(n)$
  - findKth: $O(k)$
  - insert e remove: $O(1)$

Exemplos de algoritmos anteriores

- Pilhas
  - push e pop, isEmpty: $O(1)$, tanto array como lista ligada
  - Em arrays: tempo constante e mais rápida (uma instrução da máquina apenas)
- Filas
  - Todas as operações: $O(1)$