SEMANAS:
- INICIO/APRESENTAÇÃO
- INTRODUÇÃO AOS ALGORITMOS E ESTRUTURAS DE DADOS
- ESTRUTURAS DE CONTROLO DE FLUXO EM C#
- CONCEITO DE CLASSE E OBJETO
- SEPARAÇÃO ENTRE INTERFACE PÚBLICA E IMPLEMENTAÇÃO PRIVADA
- ESTRUTURAS DE DADOS – INTRODUÇÃO
- ANÁLISE DE ALGORITMOS E COMPLEXIDADE
- ALGORITMOS DE ORDENAÇÃO
- ALGORITMOS DE ORDENAÇÃO (PARTE 2)
- PROVA DE AVALIAÇÃO
- LISTAS E FILAS
- TABELAS DE HASH
- TRATAMENTO DE COLISÕES
- ÁRVORES BINÁRIAS
- TIPOS GENÉRICOS E TRATAMENTO DE EXCEPÇÕES
- PROVA DE AVALIAÇÃO

Algoritmos e Estruturas de Dados

CTeSP - Tecnologias e Programação de Sistemas de Informação 2020/21

INSTITUTO POLITÉCNICO DA MAIA IPMAIA

Microsoft Visual Studio Professional Edition

Docente: Ricardo Henriques.

Sumário:

- Análise de Algoritmos e Complexidade
- Notação *O(n)*
  - SelectionSort - estudo para exemplo da utilização da notação O(n).

Análise de Algoritmos e Complexidade

Quase todos nós, quer a utilizar computadores, ou *smartphones* já encontramos situações em que as aplicações correram mais lento do que esperávamos, ou deixaram de responder. É normal, mais cedo ou mais tarde enquanto programadores termos de encarar questões como:

- Quanto tempo demorará o meu programa a concluir?
- Porque é que o meu programa excedeu a memória?

Em ações mais complexas como reconstruir uma biblioteca de música ou transformar um conjunto de fotografias/video, ao instalar uma aplicação, ou a trabalhar com um documento muito grande estas questões certamente vêm acima. Contudo são muito vagas para serem respondidas de forma precisa, dado que dependem de muitos factores:

- caraterísticas particulares do computador em uso;
- os dados (e as estruturas de dados) do documento em processamento;
- as ações do programa - as quais implementam um algoritmo.

Ora, o caminho para responder a estas questões passa por tomar uma postura científica e o método científico, utilizado para a modelação do mundo natural. Aplica-se a análise matemática para desenvolver modelos concisos de custo e fazem-se estudos/observações para avaliar esses modelos:

1. *Observar* as caraterísticas para proceder ao registo de métricas/medidas;
2. Construir uma *hipótese* de modelo que seja consistente com tais observações;
3. Utilizar o modelo para fazer *predição* de resultados/eventos com base na hipótese;
4. *Verificar* as previsões fazendo mais observações e confrontando com a realidade;
5. Iterar e *validar* até que a hipótese e as observações sejam compatíveis.

Modelos matemáticos

↑

Nos primeiros tempos das ciências de computação, Donald E. Knuth postulou que, apesar de todos os factores complicados para a compreensão dos tempos de execução de um programa, é possível construir um modelo matemático para descrever o tempo de qualquer programa. Segundo Knuth o tempo total de execução de um programa é determinado por dois factores principais:

- O custo de execução de cada instrução;
- A frequência da execução de cada instrução.

A primeira propriedade depende do computador, do compilador C# (ou da linguagem em uso) e do sistema operativo. A segunda depende do programa (do algoritmo) e do tamanho dos dados de entrada. Se soubermos para todas e de cada instrução do nosso programa podemos multiplicar e somar o tempo de todas as instruções, obtendo o tempo de execução do programa.

Crescimento de funções

O tempo de execução geralmente depende de um parâmetro *N* o qual é uma medida abstrata relacionada com o número de dados a processar. Os algoritmos têm tempo de execução proporcional a:

- **1** - muitas instruções para todo o programa são executadas só uma vez ou poucas vezes, não dependendo do tamanho (ou número) dos dados de entrada.
- ***log* N** - o tempo de execução é logarítmico:
  - cresce ligeiramente à medida que *N* cresce;
  - usual em algoritmos que resolvem um grande problema reduzindo-o a problemas de menor escala e resolvendo estes;
  - quando *N* duplica o *log N* aumenta, mas numa proporção bastante inferior e apenas duplica quando *N* aumenta para N^2;
- **N** - tempo de execução é linear
  - comum quando o processamento é feito para cada dado elemento de entrada;
  - situação óptima quando é necessário processar *N* dados de entrada (ou produzir *N* resultados);
- **N *log* N** - comum quando se reduz um problema em sub-problemas, se resolvem estes separadamente e se combinam as soluções
  - se *N* é 1000 *N log N* é 3000;
  - se *N* é 10000 *N log N* é 40000;
- $\mathbf{N^2}$ - tempo de execução quadrático
  - comum quando é preciso processar pares de dados de entrada;
  - prático apenas em pequenos problemas (ex: produto matriz - vector);
- $\mathbf{N^3}$ - tempo de execução cúbico (exemplo: produto de matrizes)
- $\mathbf{2^N}$ - tempo de execução exponencial
  - mau desempenho;
  - típico em soluções de força bruta
  - se *N* é 10 2^N é 1024;
  - se *N* é 100 2^N é 1267650600228229401496703205376;
- $\sqrt{\mathbf{N}}$
  - 3 se *N* é 1000
  - 6 se *N* é 1000000

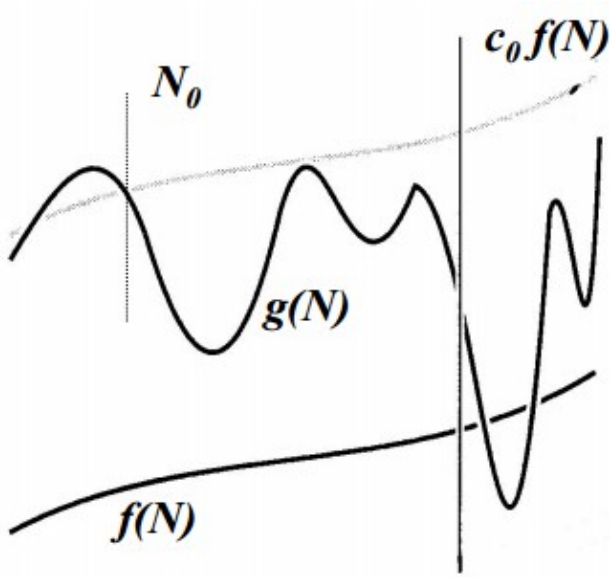
operações por segundo	tamanho do problema - 1 milhão				tamanho do problema - 1 billião		
	N	$N \lg N$	N^2		N	$N \lg N$	N^2
10^6	segundos	segundos	semanas		horas	horas	*nunca*
10^9	*instantâneo*	*instantâneo*	horas		segundos	segundos	décadas
10^{12}	*instantâneo*	*instantâneo*	segundos		*instantâneo*	*instantâneo*	semanas

Notação *O(n)* (O-grande)

A notação *O(n)* é uma notação matemática para descrever o comportamento de uma função quando esta tende para um limite e inscreve-se numa família de notações designadas de *notação Bachmann-Landau* ou *notação assimptótica* (da assimptota ou a ela relativa). Esta notação é usada nas ciências de computação para classificar algoritmos em termos do seu desempenho em medida às alterações da dimensão dos dados de entrada.

Definição

uma função g(N) diz-se ser ***O(f(N))*** *se existirem constantes* C_0 *e* N_0 *tais que* $g(N) < C_0 f(N)$ *para* $\forall\ N > N_0$



A notação permite:

- Limitar o erro que é obtido ao ignorar os termos menores das fórmulas matemáticas (do modelo de custo);
- Limitar o erro que é feito na análise ao desprezar parte de um programa que contribui de forma mínima para o custo/complexidade global;
- Classificar os algoritmos de acordo com os limites superiores no seu tempo de execução.

Os resultados da análise não são exatos, mas tecnicamente são aproximações bem caraterizadas.

Exemplo da utilização da notação *O(n)*

Vamos roubar à matéria seguinte um exemplo para percebermos a notação *O(n)*.

Algoritmo de ordenação *SelectionSort*

O que este algoritmo faz é:

1. Procura o menor elemento e troca com o elemento na 1ª posição;
2. Procura o 2º menor e coloca na 2º posição...
3. E assim sucessivamente até a ordenação estar completa.

Em código tal traduz-se em:

```
public static class SelectionSort {
    public static string[] Sort(string[] vector) {
        if (vector != null && vector.Length > 1)
            for (int i = 0; i < vector.Length; i++) {
                int min = i;
                for (int j = i + 1; j < vector.Length; j++) {
                    if (MetodosAuxiliares.Less(vector[j], vector[min]))
                        min = j;
                }
                MetodosAuxiliares.Swap(ref vector[min], ref vector[i]);
            }
        return vector;
    }
}
```

Tem-se então:

- O ciclo interno apenas faz comparações
  - Troca de elementos é feita fora do ciclo interno;
  - Cada troca coloca um elemento na sua posição final, pelo que o número de trocas é *N-1*;
  - O tempo de execução é dominado pelo número de comparações!
- Para cada item *i* (de *N-1* iterações) há uma troca e *N-i* comparações. Logo há:
  - *N-1* trocas
  - *(N-1)+(N-2)+...+2+1* comparações, ou seja aproximadamente *N(N-1)/2* comparações

Em resumo, no *Selection Sort*:

- Usa aproximadamente $\boldsymbol{N^2/2}$ comparações
- ***N*** trocas

Pelo que o desempenho é independente da ordenação inicial dos dados. Varia sim o número de vezes que o `min` é atualizado (quadrático no pior dos casos, *N log N* em média).

Considere-se então:

- t_1 - o tempo necessário para a comparação;
- t_2 - o tempo necessário para atualizar o `min`;

Tem-se que:

$T(N) = (t_1 * N^2/2) + (t_2*N)$

Como t_1 e t_2 são constantes para um dado computador, é possível encontrar um C_0 constante tal que se verifique:

$T(N) = (C_0 * N^2/2) + (C_0*N)$
ou seja
$T(N) = C_0 * (N^2/2 + N)$

Considerando $g(n) = n^2/2 + n$, então o nosso $T(N) = O(g(n))$, o qual nos dá a ideia da ordem de crescimento do tempo de execução com a variação de n.

Para diferentes computadores os tempos t_1 e t_2 são diferentes, mas $g(n)$ é constante.

MRH, 2020/21