Apontamentos de Estruturas de Dados

Tipos de Dados Abstratos e Eficiência dos Algoritmos

Ricardo Santos| rjs@estg.ipp.pt

Escola Superior de Tecnologia e Gestão
Instituto Politécnico do Porto

Última versão: Outubro de 2022

Índice

3. Estruturas de Dados

Estudamos estruturas de dados para que possamos escrever programas mais eficientes. Uma pergunta legítima que se coloca é o porquê de termos essa necessidade de os programas serem eficientes quando os computadores são mais rápidos a cada ano que passa. A resposta para essa pergunta está presente no nosso quotidiano já é que as nossas ambições crescem a par e passo com as nossas capacidades. Em vez de as necessidades de eficiência se tornarem obsoletas, a revolução moderna que envolve a transformação digital através da informatização de tarefas mais complexas trouxe um maior poder computacional e capacidade de armazenamento para que tudo isso fosse possível.

Claro que a busca pela eficiência de um programa não precisa e não deve entrar em conflito com um design sólido e uma codificação clara. A criação de programas eficientes tem pouco a ver com "truques de programação", mas sim com base na boa organização da informação e claro, bons algoritmos. Um programador que não domina os princípios básicos de algoritmia não será obviamente capaz de escrever programas eficientes. Por outro lado, a "engenharia de software" não pode ser usada como desculpa para justificar o desempenho ineficiente. A abstração no design pode e deve ser alcançada sem sacrificar o desempenho, mas isso só pode ser possível se o programador entender como medir o desempenho e fizer isso como parte integrante do processo de design e implementação.

3.1. Tipos de Dados Abstratos

Para muitos problemas, a capacidade de formular um algoritmo eficiente depende da capacidade de organizar os dados de maneira apropriada. O termo estrutura de dados é usado para denotar uma maneira particular de organizar dados para tipos específicos de operações. Iremos estudar várias estruturas de dados, desde *arrays* e listas até estruturas mais complexas, como árvores, *heaps* tabelas de *hash* e grafos, e iremos constatar como a sua escolha afeta a eficiência dos algoritmos e por sua vez das soluções desenvolvidas.

Muitas vezes, queremos falar sobre estruturas de dados sem termos que nos preocupar com todos os detalhes de implementação associados a linguagens de programação específicas ou como os dados são armazenados na memória do computador. Podemos fazer isso formulando modelos matemáticos abstratos de classes particulares de estruturas de dados ou tipos de dados que possuem características comuns. Estes são chamados de tipos de dados abstratos (*Abstract Data Type* - ADT) e são definidos apenas pelas operações que podem ser realizadas. Normalmente, especificamos como são concebidos a partir de tipos de dados mais primitivos (por exemplo, inteiros ou *strings*), e mais tarde obtemos esses dados assim como algumas verificações básicas para controlar o fluxo de processamento em algoritmos. A ideia principal é que os detalhes de implementação estão ocultos do utilizador (cliente que usa o ADT) e protegidos do acesso externo sendo conhecido como encapsulamento.

Podemos então dizer que um ADT é uma coleção de objetos de dados caracterizados pela forma como os objetos são acedidos; é um conceito humano abstrato significativo fora da ciência da computação. (Observe que "objeto", aqui, também é um conceito abstrato geral, ou seja, pode ser um "elemento" (como um inteiro), uma estrutura de dados (por exemplo, uma lista de listas) ou uma instância de uma classe (por exemplo, uma lista de círculos). Um tipo de dados é abstrato no sentido em que é independente de várias implementações concretas. ADTs comuns normalmente implementados em linguagens de programação (ou suas bibliotecas) incluem: *Arrays*, Listas, *Stacks*, *Queues*, Árvores, Tabelas de *Hash* e Grafos.

3.2. O que é uma Estrutura de Dados?

Uma estrutura de dados[1] é a implementação de um ADT numa determinada linguagem de programação. Uma estrutura de dados cuidadosamente escolhida irá permitir que o algoritmo mais eficiente seja usado. Assim, uma estrutura de dados bem projetada permite que uma variedade de operações críticas sejam realizadas usando recursos, tanto tempo de execução como espaço de memória, de modo mais eficiente.

[1] As estruturas de dados também podem ser chamadas de "*data aggregates*"

3.3. Classificação de Estruturas de Dados

As estruturas de dados são de uma forma geral divididas em:

- Estruturas de dados lineares;
- Estruturas de dados não lineares.

3.3.1. Estruturas de Dados Lineares

Estruturas de dados lineares são estruturas de dados em que os elementos de dados individuais são armazenados e acedidos de forma linear na memória do computador. Para o propósito desta unidade curricular serão estudadas as seguintes estruturas de dados lineares: listas, pilhas (*stacks*) e filas (*queues*) para determinar como a informação é processada durante a implementação.

Listas, *Queues* e *Stacks* são estruturas de dados lineares e cada uma serve como repositório em que os elementos são adicionados e removidos à vontade. Diferem umas das outras na forma como as entradas podem ser acedidas depois de adicionadas.

Lista

A lista é uma estrutura linear de entradas que podem ser adicionadas, removidas e pesquisadas sem restrições. Existem dois tipos de listas:

- Listas ordenadas;
- Listas não ordenadas (ou desordenadas).

Stack

As entradas apenas podem ser removidas de acordo com a ordem contrárias com que foram adicionadas.

- Estruturas de dados *Last-in-first-out* (LIFO);
- Não existe pesquisa por entradas na *Stack*.

Queue

As entradas apenas podem ser removidos de acordo com a ordem com que foram adicionadas

- Estruturas de dados *First-in-first-out* (FIFO);
- Não existe pesquisa por entradas na *Queue*.

3.3.2. Estruturas de Dados Não Lineares

Uma estrutura de dados não linear como o nome indica é uma estrutura de dados na qual os itens de dados não são armazenados linearmente na memória do computador, no entanto, podem ser processados usando algumas técnicas ou regras. Estruturas de dados não lineares típicas a serem estudadas nesta unidade curricular são as Árvores, Tabelas de *Hash* e os Grafos.

Árvore Geral

Modelam uma hierarquia como a estrutura de uma organização, ou uma árvore genealógica

- Organização não linear das entradas é uma generalização da estrutura em árvore binária.

Árvore Binária

É a estrutura em árvore em que cada nodo tem no máximo 2 filhos. Consiste em entradas em que cada uma delas contribui para a árvore como um todo baseando-se na sua posição na árvore.

- Mover uma entrada de uma posição para outra altera o significado da árvore binária.

Árvore Binária de Pesquisa

Possui a mesma forma estrutural de uma Árvore Binária mas cada entrada é independente: não contribui de maneira diferente se sua posição na árvore é alterada, nem a árvore como um todo possui um significado relacionado com a organização relativa das entradas.

- Organização de forma ordenada: árvore análoga à lista ordenada.

Tabela de *Hash*

Armazena as entradas com o único objectivo de permitir uma pesquisa eficiente. Requer um certo conhecimento matemático e das propriedades dos números, e das denominadas funções de *hash* que irão permitir manipular os números.

Grafos

Um grafo é um tipo especial de árvore geral já que a hierarquia é um sistema especial de relações entre as entidades. Os grafos podem ser usados para modelar sistemas de ligações físicas tais como redes de computadores, linhas aéreas, etc. e ainda relacionamentos abstractos. Depois de conseguirmos modelar um sistema através de um grafo podemos usar alguns algoritmos padrão para responder a algumas perguntas que podemos realizar aos sistemas. Existem dois tipos de grafos:

- Grafos dirigidos (também conhecidos por orientados ou direcionados) - relacionamento assimétrico
- Grafos não dirigidos - relacionamento simétrico

3.4. Exercícios Propostos

Exercício 1

O que é para si uma estrutura de dados?

Exercício 2

O que é um tipo de dados abstrato?

Exercício 3

Que tipos de estruturas de dados lineares conhece e como se diferenciam?

Exercício 4

Que tipos de estruturas de dados não-lineares conhece e como se diferenciam?

4. Análise da Eficiência dos Algoritmos

Já observámos que ao desenvolver algoritmos é importante considerar quão eficientes são para que possamos fazer escolhas informadas sobre quais os melhores para usar em determinadas circunstâncias. Portanto, antes de prosseguirmos para o estudo de estruturas de dados e algoritmos cada vez mais complexos primeiro iremos analisar com mais detalhe como medir e descrever sua eficiência.

4.1. Complexidade de Tempo versus Espaço

Ao criarmos software profissional é geralmente necessário avaliar a rapidez com que um algoritmo ou programa pode concluir as tarefas determinadas. Por exemplo, se estivermos a desenvolver um sistema de reservas de voos não será aceitável tanto para o agente de viagens como para o cliente esperarem meia hora para que uma transação seja concluída. Deve-se garantir que o tempo de espera seja razoável para o tamanho do problema e, normalmente, na execução mais rápida é melhor. Falamos sobre a complexidade de tempo do algoritmo como um indicador de como o tempo de execução depende da dimensão do conjunto de dados (*n*) de entrada a considerar.

Outra consideração importante de eficiência é quanta memória um determinado programa irá exigir para uma tarefa específica, embora com computadores modernos isso tenda a ser um problema menor do que costumava ser. Aqui falamos sobre a complexidade do espaço na forma de requisito de memória que irá obviamente depender do tamanho da estrutura de dados.

Para uma determinada tarefa ou funcionalidade existem algoritmos que trocam tempo por espaço e vice-versa. Por exemplo, se considerarmos um dispositivo de armazenamento de dados as tabelas de *hash* têm uma complexidade de tempo muito boa à custa de usar mais memória do que o necessário por outros algoritmos. Geralmente, cabe ao programador do algoritmo/programa decidir a melhor forma de equilibrar a negociação entre tempo e espaço.

4.2. Desempenho Médio versus Pior Caso

Além da negociação tempo-espaço também deve ser analisada a eficiência em termos do desempenho, isto é, se é o desempenho médio do programa que é importante, ou se é mais importante garantir que mesmo no pior caso o desempenho obedeça a certas regras. Talvez para a maior parte dos programas o caso médio é o mais importante porque economizar tempo é geralmente mais importante do que garantir um bom comportamento no pior caso. No entanto, para problemas de tempo crítico como rastrear aviões em determinados setores do espaço aéreo pode ser totalmente inaceitável que o programa demore muito se surgir o pior caso.

Os programas geralmente compensam a eficiência do caso médio com a eficiência do pior caso. Por exemplo, o algoritmo mais eficiente em média pode ter uma eficiência de pior caso.

4.3. Medidas Concretas para o Desempenho

Atualmente, estamos mais interessados na complexidade do tempo. Para isso, primeiro temos que decidir como medi-lo. Algo que se pode tentar fazer é simplesmente implementar o algoritmo, executá-lo e ver quanto tempo demora para ser executado. No entanto, essa abordagem tem vários problemas. Por um lado se o programa for muito grande e existirem vários algoritmos possíveis todos eles teriam que ser programados primeiro antes de serem comparados. Portanto, uma quantidade considerável de tempo seria desperdiçada com a implementação de programas que não seriam usados. Além disso, o computador onde o programa é executado ou mesmo o compilador usado pode influenciar o tempo de execução. Teríamos ainda que nos certificar que os dados testados são adequados para o programa. Geralmente é melhor medirmos a complexidade de uma maneira diferente. Para não ficarmos presos a uma determinada linguagem de programação ou arquitetura é melhor analisarmos a eficiência do algoritmo invés da sua implementação. Para que essa análise seja possível o algoritmo deve ser descrito de uma forma muito semelhante à sua futura implementação. É por esse motivo que normalmente os algoritmos são expressados sob a foram de pseudocódigo já que se aproxima de uma linguagem de programação.

Para determinarmos a complexidade de tempo de um algoritmo é necessário contar o número de vezes que cada operação irá ocorrer que geralmente depende do tamanho do problema. O tamanho de um problema é normalmente expresso como um número inteiro que representa o número de itens que são manipulados. Por exemplo, num algoritmo de pesquisa são o conjunto de elementos entre os quais estamos a pesquisar, num algoritmo de ordenação é o número de itens a serem ordenados. Assim, a complexidade de um algoritmo será dada através de uma função que mapeia o número de itens para o tempo que demora a processar. Esta função é denominada **função de crescimento**.

Nos primórdios dos sistemas computacionais as várias operações eram normalmente contabilizadas em proporção ao seu "custo de tempo" específico e somadas, sendo por exemplo a multiplicação de inteiros tipicamente considerada muito mais cara do que sua soma. Hoje em dia, os sistemas computacionais são muito mais rápidos e possuem inclusivamente hardware de virgula flutuante (que antigamente não era comum) tornando as diferenças de custo de tempo menos importantes. No entanto, ainda precisamos ter cuidado ao decidir considerar todas as operações como igualmente dispendiosas – aplicar alguma função, por exemplo, pode demorar muito mais do que somar simplesmente dois números sendo que as trocas geralmente demoram muito mais do que as comparações. Contabilizar apenas as operações mais caras costuma ser uma boa estratégia.

4.4. Análise Assintótica

A análise assintótica define a base matemática do desempenho em tempo de execução de um algoritmo. Se não houver qualquer *input* para um determinado algoritmo, o algoritmo irá funcionar sempre em tempo constante.

- A análise assintótica é o tempo de execução de qualquer processo ou algoritmo em termos matemáticos.
- Podemos calcular os cenários de melhor, médio e pior caso para um algoritmo através da análise assintótica.

Na programação, a análise assintótica diz-nos o comportamento geral de um algoritmo através da sua taxa de crescimento. Quanto menor o tempo de execução, melhor é o desempenho de um algoritmo. Por exemplo, vamos supor que temos que adicionar um elemento no início de um *array*. Como um *array* é uma alocação de memória contígua não podemos adicionar um elemento diretamente na primeira posição. Precisamos deslocar cada elemento para a próxima posição e só depois podemos adicionar o elemento na primeira posição. Portanto, precisamos percorrer todo o *array* uma vez. Quanto maior o tamanho do array, maior o tempo de execução.

Comparativamente se fizermos a mesma operação numa lista ligada o cenário é bastante menos complexo devido ao funcionamento da lista ligada. Podemos simplesmente criar um novo nó e apontá-lo para o primeiro nó da lista ligada existente. Desta forma adicionamos o elemento na primeira posição da lista ligada sendo esta operação muito mais fácil de executar que num *array*.

Com estes dois exemplos acabamos por comparar duas estruturas de dados diferentes e selecionamos a melhor. Quanto menor o tempo de execução, maior o desempenho.

Calcular o tempo real de execução de um processo não é viável. O tempo necessário para a execução de qualquer processo depende de tamanho *n* como já foi referido. Por exemplo, percorrer um *array* de 5 elementos irá demorar menos tempo do que percorrer um *array* de 500 elementos. Podemos observar que a complexidade do tempo depende do número de elementos. Portanto, se o tamanho do problema for '*n*', então a função de crescimento será *f(n)*.

Exemplo:

Vamos supor que a função de crescimento seria $f(n) = 8n^2 + 5n + 12$

Aqui *n* representa o número de instruções executadas.

Se $n = 1$:

Percentagem de tempo gasto devido a $8n^2 = (8/(8+5+12))*100 = 32\%$

Percentagem de tempo gasto devido a $5n = (5/(8+5+12))*100 = 20\%$

Percentagem de tempo gasto devido a $12 = (12/(8+5+12))*100 = 48\%$

No exemplo anterior podemos ver que a maior parte do tempo é relativa ao termo '12'. Mas com base em apenas um exemplo não podemos concluir a complexidade do tempo. Temos que calcular o fator de crescimento e depois observar a situação como é feito na tabela 1.

Tabela 1- Cálculo do fator de crescimento

n	***8n²***	***5n***	***12***
1	32%	20%	48%
10	92,8%	5,8%	1,4%
100	99,36%	0,62%	0,015%
1000	99,93%	0,06%	≈0%

Na tabela podemos observar que a maior relevância está no termo $8n^2$. É maior do que os outros dois termos juntos, não pode ser ignorado. Portanto, podemos afirmar que a complexidade para este algoritmo é: $f(n) = 8n^2$

Esta é a complexidade de tempo aproximada que está muito próxima do resultado real e é denominada como complexidade assintótica determinada pelo termo dominante da função de crescimento. Pode também ser referido como a ordem do algoritmo.

4.5. Notação Assintótica

As notações assintóticas mais usadas para representar a complexidade de tempo de execução de um algoritmo são:

- Notação *Big-O* (O)
- Notação Omega (Ω)
- Notação Theta (θ)

A notação usada daqui por diante será a do *Big-O (O)* que representa a ordem do algoritmo que foi calculado anteriormente e representa o pior caso. A próxima tabela apresenta outros exemplos de *Big-O* obtidos a partir de novas funções de crescimento.

Tabela 2 - Exemplos de Big-O para outras funções de crescimento

Função de Crescimento	Ordem	Descrição
f(n) = 17	O(1)	Constante
f(n) = 3log n	O(log n)	Logarítmica
f(n) = 20n - 4	O(n)	Linear
f(n) = 12n log n + 100n	O(n log n)	n log n
f(n) = $3n^2$ + 5n -2	O(n^2)	Quadrática
f(n) = $8n^3$ + $3n^2$	O(n^3)	Cúbica
f(n) = 2n + $18n^2$ + 3n	O(2^n)	Exponencial

De seguida são apresentados dois gráficos que comparam as funções de crescimento típicas para valores de *n* pequenos e n grandes. Nos gráficos o eixo do *x* representa *n* e o eixo dos *y* a complexidade do tempo de execução em termos da função de crescimento *f(n)* do Para *n* pequenos (figura 1):

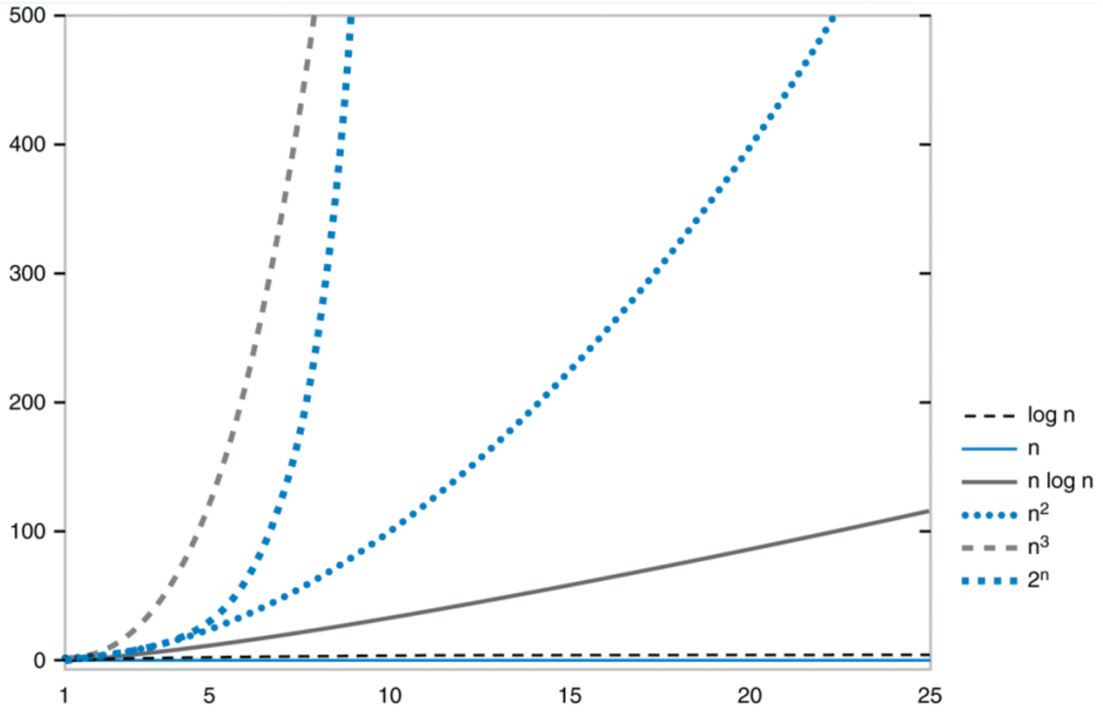

Figura 1- Função de crescimento para n pequenos

Para *n* grandes (figura 2):

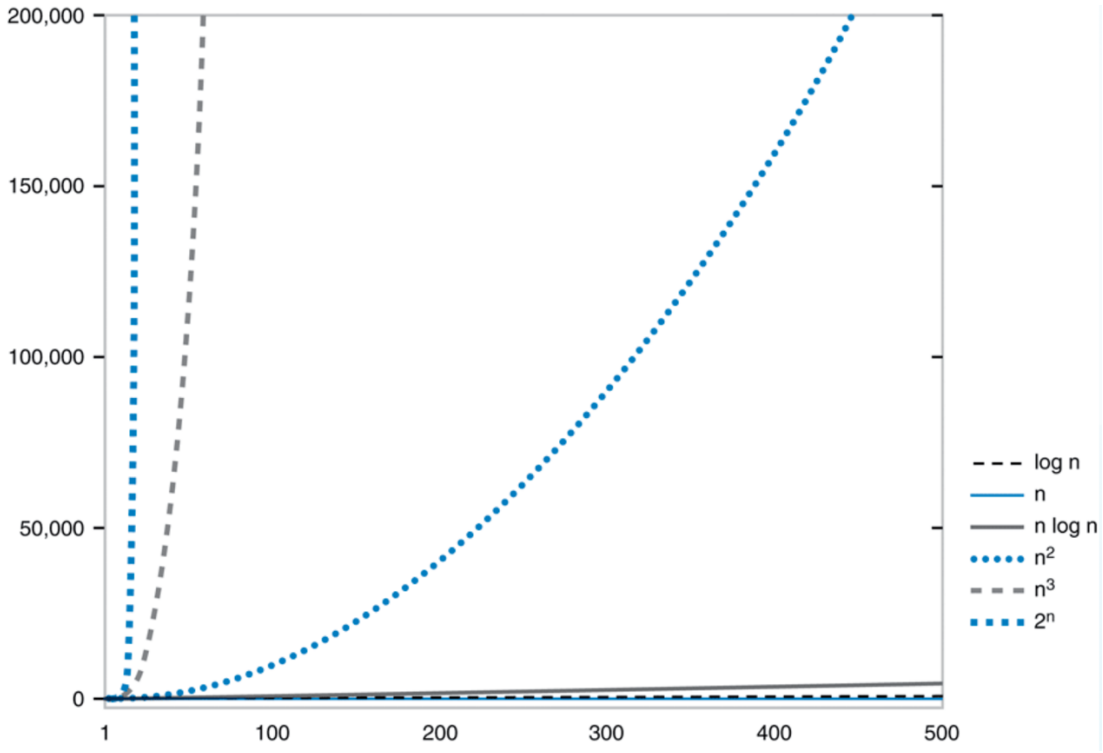

Figura 2 - Função de crescimento para n grandes

4.6. Análise de Algoritmos

Iremos usar uma abordagem muito simplista para a análise da eficiência de algoritmos que será aplicada aos ciclos e às invocações de métodos de forma a demostrar os conceitos que aprendemos.

Análise de Execução de um ciclo

Um ciclo executa um certo número de vezes (digamos *n*). Assim, a complexidade de um ciclo é *n* vezes a complexidade do corpo do ciclo. Quando os ciclos estão aninhados o corpo do ciclo exterior inclui a complexidade do ciclo aninhado (interior). Por exemplo:

```
1. for (int i=0; i<n; i++){
2.     x = x + 1;
3. }
```

O ciclo apresentado anteriormente tem uma eficiência *Big-O* de *O(n)* devido ao ciclo executar *n* vezes o corpo do ciclo que é *O(1)*. Já no exemplo seguinte:

```
1. for (int i=0; i<n; i++){
2.     x = x + 1;
3.     for (int j=0; j<n; j++){
4.        y = y - 1;
5.     }
6. }
```

Tem uma eficiência *Big-O* de $O(n^2)$ porque o ciclo é executado *n* vezes e o corpo do ciclo, que inclui um ciclo aninhado é *O(n)*.

Análise às Chamadas de Métodos

Para analisarmos as chamadas de métodos, simplesmente substituímos as chamada de métodos com a ordem do corpo do método, o exemplo apresentado de seguida é uma chamada um método com eficiência *O(1)*.

```
1. public void printsum(int count){
2.     sum = count*(count+1)/2;
3.     System.out.println(sum);
4. }
```

4.7. Exercícios Propostos

Exercício 1

Pense num programa que tenha implementado que seja inaceitavelmente lento. Identifique as operações específicas que tornam o programa lento. Identifique outras operações básicas que o programa executa com rapidez suficiente.

Exercício 2

Defina um ADT para carateres de uma *string*. O ADT deve consistir em métodos típicos que podem ser executadas em *strings* com cada método definido em termos de inputs e outputs. De seguida, defina duas representações físicas de *string*.

Exercício 4

Defina um ADT para uma matriz de inteiros. Especifique com precisão as operações básicas que podem ser executadas nessa matriz. De seguida, imagine um programa que armazena um *array* com 1.000 linhas e 1.000 colunas, onde menos de 10.000 dos valores do *array* são diferentes de zero. Descreva duas implementações diferentes para esses *arrays* que seriam mais eficientes em termos de espaço do que uma implementação em matriz que requer um milhão de posições.

Exercício 5

Comente a seguinte afirmação: *"É possível concatenar duas listas ligadas com uma eficiência de O(1)"*

Exercício 6

Porquê que a ordem de crescimento de uma função é pertinente para a análise de algoritmos?