Universidade Tecnológica Federal do Paraná (UTFPR)
Departamento Acadêmico de Informática (DAINF)
Estrutura de Dados I
Professor: Rodrigo Minetto
Exercícios (recursão)

Exercícios (seleção): necessário entregar TODOS (moodle)!

Exercício 1) Escreva uma função recursiva que recebe uma lista encadeada como entrada e retorna a soma dos elementos. Por exemplo, se $\ell = \{5, 6, 7, 8, 9, 4, 3, 2, 1, 0\}$ então o retorno da função é 45. A função deve utilizar o seguinte protótipo:

```
int sum (List *l);
```

Exercício 2) Escreva uma função recursiva que recebe uma lista encadeada e um inteiro k como entrada e retorna verdadeiro se o elemento k **pertence** a lista ou falso caso contrário. A função deve utilizar o seguinte protótipo:

```
int in (List *l, int k);
```

Exercício 3) Escreva uma função recursiva que recebe duas listas encadeadas A e B ordenadas como entrada e faz a fusão de A e B, mantendo-a ordenada. Não devem ser alocados (criados) nós extras. Os nós serão religados para compor a nova lista C ordenada. Por exemplo, se $A = \{0, 2, 4, 6, 8\}$ e $B = \{1, 3, 3, 5\}$, então $C = \{0, 1, 2, 3, 3, 4, 5, 6, 8\}$, tal que C tem os nós de A e B, então para desalocar memória basta liberar C. A função deve utilizar o seguinte protótipo:

```
List* merge (List *A, List *B);
```

Exercício 4) Escreva uma função **recursiva** que recebe uma lista encadeada como entrada e retorna o **tamanho** da lista. Por exemplo, se $\ell = \{5, 6, 7, 8, 9, 4, 3, 2, 1, 0\}$ então o retorno da função é 10. A função deve utilizar o seguinte protótipo:

```
int size (List *l);
```

A estrutura de dados e operações básicas estão em anexo ao material da aula (**arquivos.zip**) para auxiliar.

Exercícios (aprofundamento): não é necessário entregar mas é importante estudar!

Exercício 5) Escreva uma função recursiva que recebe uma lista encadeada como entrada e **libera** (destrói) todos os elementos armazenados. A função deve utilizar o seguinte protótipo:

```
void destroy (List *l);
```

Exercício 6) Escreva uma função recursiva que recebe uma lista encadeada como entrada e retorna o **maior** elemento armazenado. Por exemplo, se $\ell = \{5, 6, 7, 8, 9, 4, 3, 2, 1, 0\}$ então o retorno da função é 9. A função deve utilizar o seguinte protótipo:

```
int max (List *l);
```

Exercício 7) Escreva uma função recursiva que recebe uma lista encadeada e um inteiro k como entrada e retorna a posição da primeira ocorrência de k na lista: 0 se estiver no primeiro nó, 1 se estiver no segundo, ..., ou -1 se ele não existe. Por exemplo, se $\ell = \{5, 6, 7, 8, 9, 4, 3, 2, 1, 0\}$ e $k = 7$, então o retorno da função é 2. A função deve utilizar o seguinte protótipo:

```
int position (List *l, int k);
```

Exercício 8) Escreva uma função recursiva que recebe uma lista encadeada e um inteiro k como entrada e remove a primeira ocorrência de k na lista. Por exemplo, se $\ell = \{0, 1, 2, 1\}$ e $k = 1$, então após a remoção lista = $\ell = \{0, 2, 1\}$. A função deve utilizar o seguinte protótipo:

```
List* removek (List *l, int k);
```

Ps. se o elemento não existir então a lista deve permanecer inalterada. Teste os limites da lista para ter certeza que sua função funciona, por exemplo, remover a cabeça, meio e cauda da lista.

Exercício 9) Escreva uma função recursiva que recebe uma lista encadeada e um inteiro k como entrada e remove todas as ocorrências de k na lista. Por exemplo, se $\ell = \{1, 0, 1, 2, 1\}$ e $k = 1$, então após a remoção $\ell = \{0, 2\}$. A função deve utilizar o seguinte protótipo:

```
List* remove_all (List *l, int k);
```

Ps. se o elemento não existir então a lista deve permanecer inalterada. Teste os limites da lista para ter certeza que sua função funciona, por exemplo, remover a cabeça, meio e cauda da lista.

Exercício 10) Escreva uma função recursiva que recebe uma lista encadeada e um inteiro k como entrada e insere k na lista respeitando a ordenação. Por exemplo, se os valores de k sucessivos são $9, 0, 4, 2, 7$, então $l = \{0, 2, 4, 7, 9\}$. A função deve utilizar o seguinte protótipo:

```
List* insert_sort (List *l, int k);
```

Exercício 11) Escreva uma função recursiva que recebe uma lista encadeada A como entrada e realiza uma cópia, ou seja, para cada nó em A realize a alocação de um novo nó em uma nova lista B. Para ter certeza que a cópia foi realizada de forma correta, destrua a lista A e após imprima o conteúdo de B. A função deve utilizar o seguinte protótipo:

```
List* copy (List *A);
```

Exercício 12) Escreva uma função recursiva que recebe duas listas encadeadas A e B como entrada e retorna true se elas são iguais, ou falso caso contrário. Para duas listas serem iguais elas precisam ter os elementos na mesma ordem e os mesmos valores. A função deve utilizar o seguinte protótipo:

```
int similar (List *A, List *B);
```

Exercício 13) Escreva uma função recursiva que recebe duas listas encadeadas A e B como entrada e cria um novo conjunto C de intersecção entre elas. Por exemplo, se $A = \{0, 5, 10, 15, 20, 25, 30\}$ e $B = \{0, 3, 6, 9, 12, 15, 18, 21, 24, 27, 30\}$, então o conjunto intersecção é $C = \{0, 15, 30\}$. A função deve utilizar o seguinte protótipo:

```
List* intersection (List *A, List *B);
```

Dica: use a função recursiva **pertence** nessa solução.

Exercício 14) Escreva uma função recursiva que recebe uma lista encadeada como entrada e imprime a lista em ordem reversa. Por exemplo, se $\ell = \{1, 2, 3, 4, 5\}$ então o retorno da função é $\{5, 4, 3, 2, 1\}$ (ordem de impressão, ou seja, não modificar a lista). A função deve utilizar o seguinte protótipo:

```
void print_reverse (List *l);
```

Ps. não use vetores ou espaço auxiliar apenas se baseie na lista de entrada e recursão.

⋆ **Exercício 15 (explora recursão mas não listas) Sudoku** é um quebra-cabeça baseado na colocação lógica de números. O objetivo do jogo é a colocação de números de 1 a 9 (números em vermelho na figura abaixo) em cada uma das células vazias numa grade de 9×9, constituída por 3×3 subgrades chamadas regiões. O quebra-cabeça contém algumas pistas iniciais, que são números inseridos em algumas células (na cor preta na figura abaixo), de maneira a permitir uma dedução dos números em células que estejam vazias. Cada coluna, linha e subgrade só pode ter **UM** número de cada tipo:

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

Uma estratégia para resolver o quebra-cabeça acima é visitar todas as células vazias, preenchendo com números válidos, e caso o algoritmo encontre alguma inconformidade ao testar uma possibilidade, então retroceda e avalie outra, técnica esta conhecida como **backtracking**. A vantagem da técnica por **backtracking** é que ela acha uma solução, se alguma existir, com a desvantagem do custo computacional — estratégia por força bruta se não houver alguma limitação ao explorar o espaço de busca.

Se tiver interesse em resolver esse quebra-cabeça, implemente a funcionalidade que falta conforme indicado no programa **sudoku.c** (em **arquivos.zip**).