6.1 Defina uma função `int identidade(int mat[N][N])` para verificar se uma matrix $N \times N$ é a *identidade* (isto é, contém 1 na diagional principal e 0 nas outras posições).

O resultado deve ser 1 ou 0 conforme a matriz é ou não identidade. A sua função deve funcionar para qualquer dimensão N declarada como constante usando usando `#define`.

6.2 Defina duas funções

```
int soma_diagonal1(int mat[N][N]);
int soma_diagonal2(int mat[N][N]);
```

que dadas matrizes $N \times N$ soma: (1) os valores na diagonal principal (isto, tais que índices i, j são iguais); (2) os valores na diagonal secundária (isto é, tais que os índices satisfazem $i + j = N - 1$). As funções devem funcionar para qualquer dimensão N declarada como constante usando `#define`.

6.3 Um *quadrado mágico* é uma matriz quadrada de números inteiros tal que todas as linhas, colunas e diagonais somam o mesmo valor. No exemplo seguinte cada linha, coluna e diagonal soma o mesmo valor (15 neste caso):

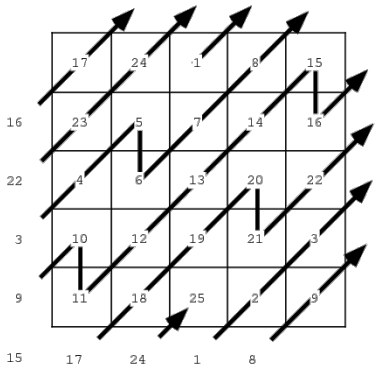
$$\begin{bmatrix} 2 & 7 & 6 \\ 9 & 5 & 1 \\ 4 & 3 & 8 \end{bmatrix}$$

Escreva uma função `int magico(int a[20][20], int n)` que testa se uma matriz é um quadrado mágico. A matriz é representada por uma variável indexada `a` com dimensão declarada 20×20; o argumento `n` indica qual sub-matriz a considerar: por exemplo, se `n = 3` devemos testar se a sub-matriz de 3×3 é quadrado mágico (e ignorar o resto da matriz). O resultado deve ser um inteiro: 1 se é quadrado mágico e 0 caso contrário.

6.4

O algoritmo seguinte permite gerar quadrados mágicos de tamanho $n \times n$ para n ímpar: começemos por colocar um "1" no meio da primeira linha; em seguida, vamos colocar os números de 2 a n^2: o próximo numero é colocado na posição na diagonal para cima e para a direita se esta se encontrar vazia; se a posição já se encontra preenchida, então passamos para a posição em baixo. Em qualquer dos casos, o movimento é feito considerando que os bordos cima/baixo e esquerdo/direito do quadrado estão ligados (isto é, se sairmos pela direita re-entramos na posição correspondente à esquerda e se sairmos por cima re-entramos na posição corresponde por baixo).

A figura seguinte ilustra a geração de um quadrado mágico 5×5.



Escreva um programa que constroi um quadrado mágico de dimensão impar < 50 usando este algoritmo.

6.5 Sejam `i` uma variável inteira e `p` e `q` apontadores para inteiros. Indique quais das seguintes atribuições são legais.

(a) `p = i;`
(b) `*p = i;`
(c) `p = &q;`
(d) `p = &i;`
(e) `&i = p;`
(f) `p = q;`
(g) `p = *q;`
(h) `*p = q;`
(i) `*p = *q;`

6.6 Sejam `i` uma variável `int` e `p` um apontador para `i`. Indique quais das seguintes instruções de leitura e escrita são corretas.

(a) `printf("%d",i);`
(b) `scanf("%d",i);`
(c) `scanf("%d",&i);`
(d) `printf("%d",p);`
(e) `printf("%d",*p);`
(f) `scanf("%d",p);`
(g) `printf("%p",p);`
(h) `printf("%p",i);`
(i) `printf("%p",*p);`

6.7 Escreva uma função

```
void decompor(int total_seg, int *horas, int *mins, int *segs);
```

que decompõe um total inteiro de segundos `total_seg` em horas, minutos (0–59) e segundos (0–59); os resultados devem ser atribuidos ao conteúdo dos apontadores `horas`, `mins` e `segs`. Pode assumir que o total de segundos é maior que zero.

6.8 Escreva uma função

```
void max_min(int vec[], int size, int *pmax, int *pmin);
```

que determina o valor máximo e mínimo de um vector; os resultados devem ser atribuidos ao conteúdo dos apontadores `pmax` e `pmin`. Pode assumir que `size` é sempre maior que zero.

6.9 Escreva uma função `void reduzir(int *pnum, int *pdenom)` que reduz uma fração de numerador e denominador `*pnum` e `*pdenom` à forma simplificada. O numerador e denominador devem ser modificados por meio dos apontadores `pnum` e `pdenom`. Pode assumir que o denominador é sempre diferente de zero.

Sugestão: para simplificar uma fração basta dividir o numerador e o denominador pelo seu máximo divisor comum (pode usar o algoritmo de Euclides apresentado na aula 7 para o calcular).

6.10 Re-escreva a função seguinte para usar apontadores em vez de índices (ou seja: eliminar a variável `i` e os usos de indexação `vec[...]`).

```
void store_zeros(int vec[], int n)  {
  int i;
  for(i = 0; i<n; i++)
    vec[i] = 0;
}
```

6.11 Considere a função apresentada nas aulas teóricas para contar espaços de uma cadeia de caraters. Prentende-se que escreva uma versão alternativa desta função usando apontadores em vez de índices. A função deve ter a declaração

```
int contar_espacos(char *str);
```

O resultado deve ser o número de espaços na cadeia.

6.12 Considere a função apresentada nas aulas teóricas para inverter a ordem de carateres uma cadeia. Prentende-se que escreva uma versão alternativa desta função usando apontadores em vez de índices. A função deve ter a declaração `void inverter(char *str);`

6.13 Considere a função apresentada nas aulas teóricas para procurar um carater numa cadeia. Prentende-se que escreva uma versão alternativa desta função usando apontadores em vez de índices. A função deve ter a declaração

```
char *procurar(char *str, char ch);
```

O resultado deve ser um apontador para a primeira ocorrência do carater `ch` (se este ocorrer) ou `NULL` caso contrário.

6.14 Considere a função apresentada nas aulas teóricas para comparar igualdade de cadeias de carateres. Prentende-se que escreva uma versão alternativa desta função usando apontadores em vez de índices. A função deve ter a declaração

```
int comparar(char *str1, char *str2);
```

O resultado deve ser 1 se as cadeias são iguais e 0 se são diferentes.