Fundamentos de Programação
Flags

Dainf - UTFPR

Profa. Leyza B. Dorini
Prof. Bogdan T. Nassu

Um exemplo clássico

Considere o seguinte exemplo clássico da computação

Escreva um programa que diz se um número inteiro positivo n dado é primo.

Lembrando: um número positivo é primo se for divisível somente por 1 e por ele mesmo. Os 10 primeiros números primos são: 2, 3, 5, 7, 11, 13, 17, 19, 23, 29.

Como seria um algoritmo para resolver este problema? [1]

[1] O teste de primalidade é considerado computacionalmente complicado, o que tem implicações em áreas como criptografia.

Um exemplo clássico

Considere o seguinte exemplo clássico da computação

Escreva um programa que diz se um número inteiro positivo n dado é primo.

Lembrando: um número positivo é primo se for divisível somente por 1 e por ele mesmo. Os 10 primeiros números primos são: 2, 3, 5, 7, 11, 13, 17, 19, 23, 29.

Como seria um algoritmo para resolver este problema? [1]

A solução mais comum consiste em em testar todos os divisores possíveis a partir de 2 (porque todo número é divisível por 1). Se encontrarmos algum divisor menor que n, o número não é primo.

[1] O teste de primalidade é considerado computacionalmente complicado, o que tem implicações em áreas como criptografia.

Elaborando o algoritmo

Um algoritmo simples é:

- Ler um número e armazenar na variável `n`.
- Testar em um *loop* se algum dos números entre 2 e (`n-1`) é divisor de `n`.
  - Se pelo menos um deles é divisor de `n`, então `n` não é primo.

Note que, na prática, estamos verificando se `n` **não** é primo.

Este algoritmo pode ser melhorado de algumas formas: após testar o 2, não precisaríamos testar nenhum fator par (todo número par também é divisível por 2), e não precisamos testar até (`n-1`), mas somente até (*log* (`n`)). Este tipo de otimização **não é** nosso foco, então vamos ficar com o algoritmo mais simples!

Mostrando mensagem quando n **não é primo**

```
1       int main ()
2       {
3          int div, n;
4
5          scanf("%d", &n);
6
7          for (div = 2; div < n; div++)
8          {
9             if (n%div == 0) // n é divisível por div?
10               printf ("Nao eh primo.\n");
11         }
12      }
```

O programa acima mostra uma mensagem quando n não é primo.

- Observe que o *loop* é determinado, pois sabemos o início e o final do intervalo de divisores que precisam ser testados.
- O programa imprime a mensagem a **cada** divisor encontrado!

Complementando: mensagem quando `n` é primo

Considere o seguinte exercício:

Complementar o algoritmo anterior para imprimir também uma mensagem quando `n` **é primo**.

Considerando a definição de que um número é primo se seus únicos divisores são 1 e ele mesmo:

- Ler um número e armazenar na variável `n`.
- Fazer um laço de repetição para testar se algum dos números entre 2 e (`n - 1`) é divisor de `n`.
  - Se pelo menos um deles é divisor de `n`, então `n` não é primo.
  - **Se nenhum número nesse intervalo é divisor, `n` é primo.**

E para o caso em que n é primo?

Em um primeiro momento, pode-se sugerir a seguinte solução:

```
1 for (div=2; div < n; div++)
2 {
3    if (n%div  == 0)
4       printf ("Nao eh primo.\n");
5    else
6       printf ("Eh primo.\n");
7 }
```

Entretanto, para `n = 6`, o programa irá imprimir 2 vezes `Nao eh primo` (para `div` igual a 2 e 3), e 2 vezes `Eh primo` (para `div` igual a 4 e 5).

Onde está o erro? No uso do `else`!

No algoritmo anterior, fazemos um laço de repetição para testar se algum dos números entre 2 e (`n - 1`) é divisor de `n`.

- Se pelo menos um deles é divisor de `n`, então `n` não é primo.
- Se nenhum número nesse intervalo é divisor, `n` é primo.

Veja que, para um número **não ser primo**, basta que **um** valor no intervalo seja seu divisor. Portanto, é adequado (mas muito desajeitado, dadas as mensagens repetidas) fazer:

```
se (n % div ==0) entao mostre: 'nao eh primo'
```

Por outro lado, para o número ser primo, **nenhum** valor no intervalo pode ser divisor. É por este motivo que não podemos usar o `else`: não é suficiente que apenas um valor não seja divisor!

Onde está o erro? No uso do `else`!

No algoritmo anterior, fazemos um laço de repetição para testar se algum dos números entre 2 e (`n - 1`) é divisor de `n`.

- Se pelo menos um deles é divisor de `n`, então `n` não é primo.
- Se nenhum número nesse intervalo é divisor, `n` é primo.

Veja que, para um número **não ser primo**, basta que **um** valor no intervalo seja seu divisor. Portanto, é adequado (mas muito desajeitado, dadas as mensagens repetidas) fazer:

```
se (n % div ==0) entao mostre: 'nao eh primo'
```

Por outro lado, para o número ser primo, **nenhum** valor no intervalo pode ser divisor. É por este motivo que não podemos usar o `else`: não é suficiente que apenas um valor não seja divisor!

Onde está o erro? No uso do `else`!

No algoritmo anterior, fazemos um laço de repetição para testar se algum dos números entre 2 e (`n - 1`) é divisor de `n`.

- Se pelo menos um deles é divisor de `n`, então `n` não é primo.
- Se nenhum número nesse intervalo é divisor, `n` é primo.

Veja que, para um número **não ser primo**, basta que **um** valor no intervalo seja seu divisor. Portanto, é adequado (mas muito desajeitado, dadas as mensagens repetidas) fazer:

```
se (n % div ==0) entao mostre: 'nao eh primo'
```

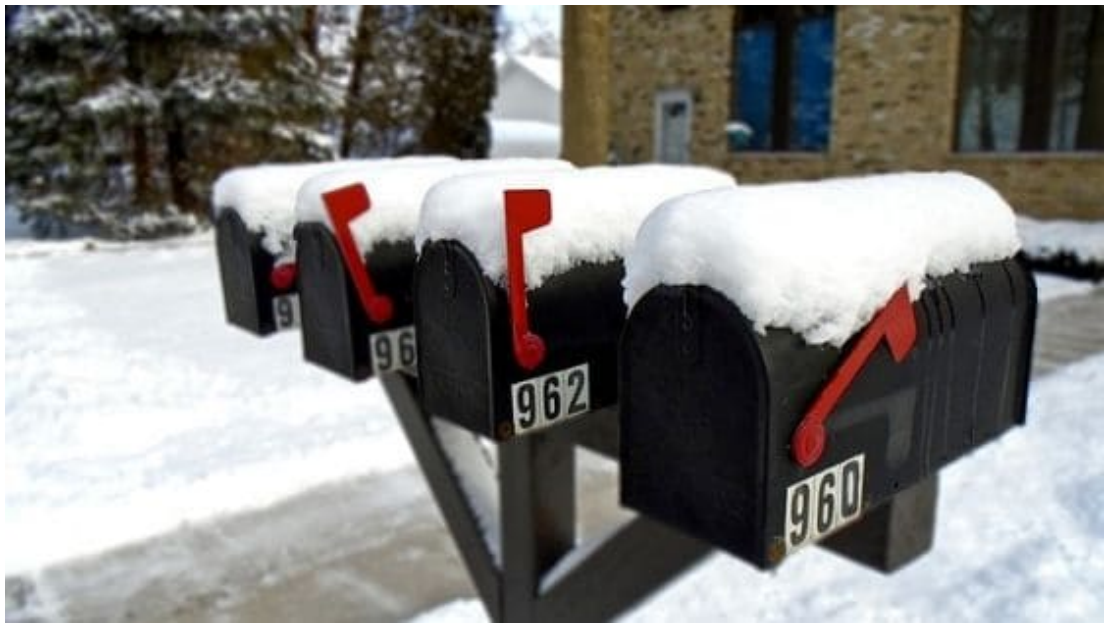
Por outro lado, para o número ser primo, **nenhum** valor no intervalo pode ser divisor. É por este motivo que não podemos usar o `else`: não é suficiente que apenas um valor não seja divisor!

Controle de laços

E agora? O que fazer nestes casos?

Vamos aprender mais uma “ferramenta”!

Variáveis indicadoras (*flags*)



Vamos criar variáveis com a caraterística das “bandeirinhas” das caixas de correio: “ativadas” ou “desativadas”.

Flags

Uma *flag* é um valor — normalmente uma variável — que sinaliza se determinado evento ocorreu ou não, ou se determinada propriedade foi ou não satisfeita.

Flags **não são** uma estrutura da linguagem C, ou algo com uma sintaxe própria (ex: `if`, `while`), e sim um padrão de uso, uma **forma** de resolver problemas (ex: contador de interações).

Ao utilizar uma *flag*, usualmente temos as seguintes etapas:

- A variável é interpretada como um valor booleano, sendo inicializada com um valor que indica se a propriedade que queremos verificar é válida (1) ou não (0).
- No decorrer do programa, verificamos se a propriedade é válida para todos os casos que precisam ser testados.
- Se em algum momento a validade não é verificada, o status da variável indicadora é invertido (de 1 para 0 ou de 0 para 1).

Número primo: solução usando *flag*

```
1       // Note que o nome da flag indica a propriedade.
2       int n, div, eh_primo;
3
4       scanf("%d", &n);
5
6       eh_primo = 1; // Supõe que n é primo por padrão.
7       for (div = 2; div < n; div++)
8           if (n%div  == 0)
9               eh_primo = 0; // Achou, inverte a flag.
10
11      // Agora, basta verificar o status da flag.
12      // Note como o if está montado.
13      if (eh_primo)
14          printf ("Eh primo.\n");
15      else
16          printf ("Nao eh primo.");
17
```

Flags

Um ponto importante a ser observado é o seguinte:

Ao testar um evento em particular, a partir do momento que uma *flag* tem o seu status alterado dentro do bloco de repetição, ela não retorna mais ao valor original!

No exemplo do número primo, o status da *flag* é alterado quando o primeiro divisor é encontrado (em outras palavras, a *flag* passa a sinalizar que o valor testado **não é primo**). Como a partir desse momento o valor não pode "voltar a ser primo", não faz sentido que a *flag* retorne ao status anterior.

Flags

Um ponto importante a ser observado é o seguinte:

Ao testar um evento em particular, a partir do momento que uma *flag* tem o seu status alterado dentro do bloco de repetição, ela não retorna mais ao valor original!

No exemplo do número primo, o status da *flag* é alterado quando o primeiro divisor é encontrado (em outras palavras, a *flag* passa a sinalizar que o valor testado **não é primo**). Como a partir desse momento o valor não pode "voltar a ser primo", não faz sentido que a *flag* retorne ao status anterior.

Portanto, **não se deve** usar `else` neste contexto

```
eh_primo = 1;
for (div=2; div < n; div++)
{
    if (n%div  == 0)
        eh_primo = 0;
    else            <<<<<< ERRADO!!!!
        eh_primo = 1; <<<<<< ERRADO!!!!
}
```

Lembre-se: depois que a *flag* é invertida, ela só retorna ao valor original se for para testar uma nova entrada.

Flags na condição do *loop*

```
1       eh_primo = 1;
2       for (div = 2; div < n; div++)
3           if (n%div  == 0)
4               eh_primo = 0;
```

No exemplo acima, a *flag* é invertida logo que encontramos **o primeiro** divisor de `n`.

- Então, **todos** os testes realizados após a *flag* se tornar 0 são inúteis!!!

Não poderíamos usar a *flag* para interromper o *loop* tão logo saibamos que o número não é primo?

Flags na condição do *loop*

Lembre-se que **qualquer** expressão que pode ser verdadeira ou falsa pode ser usada na condição do `while`...

- `while (foo == bar)`: repete enquanto `foo` for igual a `bar`.
- `while (foo)`: repete enquanto `foo != 0`.
- `while (1)`: *loop* infinito!

Então, podemos também usar expressões lógicas para incluir a *flag* na condição de um *loop*:

No exemplo do número primo: devemos repetir enquanto não tivermos testado todos os divisores entre 2 e `(n-1)`, **e** enquanto o valor da *flag* não tiver mudado para 0.

Número primo: solução melhor usando *flag*

```
1       // Note que o nome da flag indica a propriedade.
2       int n, div, eh_primo;
3
4       scanf("%d", &n);
5
6       eh_primo = 1; // Supõe que n é primo por padrão.
7       for (div = 2; div < n && eh_primo; div++)
8           if (n%div  == 0)
9               eh_primo = 0; // Achou, inverte a flag.
10
11      // Agora, basta verificar o status da flag.
12      // Note como o if está montado.
13      if (eh_primo)
14          printf ("Eh primo.\n");
15      else
16          printf ("Nao eh primo.");
17
```

Como saber quando usar *flags*?

Lembre-se!

Contadores, *flags* e acumuladores são padrões comuns, úteis em várias situações, mas eles podem aparecer juntos, ou não fazerem parte da solução de um problema. Não existem fórmulas para a criação de soluções para problemas! Em outras palavras, não existe um algoritmo para criar algoritmos.

Portanto, é importante que você siga praticando! Faça os exercícios nas listas!