06

Pensando em outros possíveis erros

Transcrição

Vamos ver qual é o outro erro? Novamente, olharemos nosso código linha a linha. Nós reenviamos o programa ao cliente depois de ler só duas linhas de código. Tinha ainda muitas outras que poderiam dar erro. Qual a chance de estar tudo ok? Poderia muito bem ter mais um erro que passou. Fomos apressados demais e agora daremos uma olhada mais cuidadosa.

Se você manda algo para o cliente dizendo que está pronto, e não funciona, vai deixá-lo chateado. Mas corrigir, reenviar e ainda sim haver erros, vai deixá-lo possesso. Você terá mentido para ele, mesmo sem saber. Mas o cliente te contratou justamente para saber. Seu trabalho é antecipar os problemas. Claro que temos uma margem de erro. Mas a senioridade de um programador vem de ficar esperto a algumas práticas cotidianas. Quando precisar buscar um erro, procure em tudo. Ele pode estar em qualquer lugar, e, em geral, está em todos os lugares.

Novamente, vamos para a análise linha a linha.

```
public static void main(String[] args){

    Scanner scanner = new Scanner(System.in);
    while(scanner.hasNext()){
      int menor = scanner.nextInt();
      int maior = scanner.nextInt();

      int maiorCicloAteAgora = 1;
      for(int atual = menor; atual <= maior; atual++){
        int resultado = calculaPara(atual);
        if(resultado > maiorCicloAteAgora){
          maiorCicloAteAgora = resultado;
        }

      }
      System.out.println(i + " " + j + maiorCicloAteAgora);
    }

    private static int calculaPara(int i){
      int impressos = 1;
      //System.out.println(n);
      while (n != 1) {
              if(n % 2 == 1) n = n * 3 + 1;
            else n = n / 2;
               impressos++
              //System.out.println(n);
      }
      return impressos;
    }
```

Começamos escaneando, e como estamos em um ambiente controlado ele sempre lerá do arquivo para o qual estamos redirecionando. Em seguida, lemos o próximo elemento (`hasNext`). Se tiver um próximo elemento, leremos o primeiro e o segundo inteiros, sendo o primeiro o `menor` e o segundo o `maior` . Está um pouco estranho, mas vamos seguir.

Definimos que o maior ciclo é `1`. Temos que pensar se esse número é realmente válido. Sabemos que se começarmos com `1`, imprimiremos pelo menos o `1`, e qualquer outro número será impresso primeiro, antes de passar pelo algoritmo e imprimir outro. Então `1` é sempre o mínimo.

Então, fazemos um `for()` do `menor` até o `maior`. Teu coração também dói quando usamos esses nomes de variáveis? Qual era o nome original dessas variáveis? No enunciado, elas são chamadas de `i` e `j`. Mudamos para deixar um pouco mais claro, mas vamos voltar aos nomes originais. Para renomear as variáveis, usamos `Ctrl + 2 + R`

```
public static void main(String[] args){

    Scanner scanner = new Scanner(System.in);
    while(scanner.hasNext()){
      int i = scanner.nextInt();
      int j = scanner.nextInt();

      int maiorCicloAteAgora = 1;
      for(int atual = i; atual <= j; atual++){
        int resultado = calculaPara(atual);
        if(resultado > maiorCicloAteAgora){
          maiorCicloAteAgora = resultado;
        }

      }
      System.out.println(i + " " + j + maiorCicloAteAgora);
    }

    private static int calculaPara(int i){
      int impressos = 1;
      //System.out.println(n);
      while (n != 1) {
              if(n % 2 == 1) n = n * 3 + 1;
            else n = n / 2;
               impressos++
              //System.out.println(n);
      }
      return impressos;
    }
```

De onde tiramos que `i` é o menor e `j` o maior? Não é só porque os exemplos dados pelo cliente começavam com o menor número que será sempre assim. O input é um par de números inteiros por linha. Sabemos apenas que eles estarão entre `0` e `1000000`. O enunciado não especifica mais nada. Os dois poderiam ser iguais? Como não há nada que diga o contrário, podem ser iguais.

Não paramos para pensar em todas as condições possíveis. Inclusive nos iludimos ao renomear as variáveis assumindo que um número era maior que o outro. Temos que ter certeza do que está escrito sempre que assumimos alguma coisa. Na dúvida, basta perguntar para o cliente, a quem temos acesso no dia a dia. Em uma maratona, isso não é possível. Temos que ler com atenção e entender de verdade o que está escrito. Em geral, a informação está toda lá, e nós que por vezes não a encontramos.

Vamos criar a `entrada4.txt`, que começará como a `entrada1.txt`, mas terá uma inversão da primeira linha ao final.

```
1 10
100 200
```

```
201 210
900 1000
10 1
```

Vamos rodar no terminal:

```
Alura-Azul:bin alura$ time java Main < entrada4.txt
1 10 20
100 200 125
201 210 89
900 1000 174
10 1 1

real    0m0.545s
user    0m0.120s
sys     0m0.034s
```

Ele imprimiu `1` para o último caso, quando deveria ser vinte. Ele fez isso porque entrou no laço `for(int atual = i; atual <= j; atual++)`, e como `10` não é menor que `1`, ele parou. Para que isso não aconteça novamente, há várias soluções. Testaremos uma delas, que pede para trocar `i` e `j`, quando o primeiro for maior.

```
public static void main(String[] args){

    Scanner scanner = new Scanner(System.in);
    while(scanner.hasNext()){
      int i = scanner.nextInt();
      int j = scanner.nextInt();

      if( i > j){
        int temporario = j;
        j = i;
        i = temporario;
      }

      int maiorCicloAteAgora = 1;
      for(int atual = i; atual <= j; atual++){
        int resultado = calculaPara(atual);
        if(resultado > maiorCicloAteAgora){
          maiorCicloAteAgora = resultado;
        }

  ...

}
```

Vamos testar?

```
Alura-Azul:bin alura$ time java Main < entrada4.txt
1 10 20
100 200 125
201 210 89
900 1000 174
1 10 20
```

```
real    0m0.140s
user    0m0.124s
sys     0m0.032s
```

Para a última linha, ele imprimiu `1 10 20`, mudando a ordem que estava na entrada. Precisamos que saia na ordem `i`, `j` e maior *cicle lenght*. Está escrito no enunciado, e devemos respeitar a ordem do input. Estamos cada vez mais atentos aos detalhes, e para cada um deles devemos criar um caso de teste. Note que fica um pouco chato ficar procurando a linha em que está o caso específico. Na vida real costumamos fazer testes de unidade, que são ensinados em outros cursos da Alura. Em uma maratona o que podemos fazer? Criamos um arquivo de saída, que se chamará aqui `saida4.txt`, que corresponderá ao que queremos que a `entrada4.txt` gera.

```
1 10 20
100 200 125
201 210 89
900 1000 174
10 1 20
```

Sabemos que a saída do `entrada4.txt` deve ser igual ao que está no arquivo `saida4.txt`. Temos como pedir para o terminal comparar os dois, com o `diff`, da seguinte maneira:

```
Alura-Azul:bin alura$ java Main < entrada4.txt | dff - saida4.txt
5c5
< 1 10 20
---
> 10 1 20
\ No newline at end of file
```

O terminal nos indica que o problema está na linha 5 (`5c5`), depois essa linha de saída `entrada4.txt` e depois como deveria ter saído (como está na `saida4.txt`). Assim, em uma maratona de programação já criamos de cara arquivos de saída esperada, e vamos usando o `diff` para ver o que deu errado.

Bom, o jeito que encontramos para resolver o problema torna a saída inadequada. Teremos que criar outra solução. Queremos começar no menor número, e terminar no maior. E o Java sabe discriminar isso, com o `Math.min(a, b)` e o `Math.max(a, b)`, que selecionam respectivamente o menor e o maior número.

```
public static void main(String[] args){

    Scanner scanner = new Scanner(System.in);
    while(scanner.hasNext()){
      int i = scanner.nextInt();
      int j = scanner.nextInt();

      if( i > j){
        int temporario = j;
        j = i;
        i = temporario;
      }

      int maiorCicloAteAgora = 1;
      for(int atual = Math.min(i, j); atual <= Math.max(i, j); atual++){
```

```
            int resultado = calculaPara(atual);
            if(resultado > maiorCicloAteAgora){
              maiorCicloAteAgora = resultado;
            }

    ...

  }
```

Vamos testar no terminal? Usaremos o `diff` novamente.

```
Alura-Azul:bin alura$ java Main < entrada4.txt | dff - saida4.txt
5c5
< 10 1 20
---
> 10 1 20
\ No newline at end of file
```

Ele nos avisa que não tem quebra de linha no final do arquivo, na linha 5. Isso acontece porque não colocamos uma quebra de linha ao final da `saida4.txt` , como havia na `entrada4.txt` . Não estamos nos preocupando muito com isso porque no enunciado é pedido para que imprimamos uma linha com os números para cada linha de input. Acrescentaremos essa linha na `saida4.txt` , porque será uma linha correspondente à linha sem nada da entrada.

```
1 10 20
100 200 125
201 210 89
900 1000 174
10 1 20
```

Vamos testar novamente o `diff` no terminal.

```
Alura-Azul:bin alura$ java Main < entrada4.txt | dff - saida4.txt
Alura-Azul:bin alura$
```

Ele não nos retorna nada, o que prova que as saídas eem questão estão iguais. Eu costumo rodar tanto com `diff` como sem ele, para ter certeza de que está saindo o que queríamos.

```
Alura-Azul:bin alura$ java Main < entrada4.txt | dff - saida4.txt
Alura-Azul:bin alura$ java Main < entrada4.txt
1 10 20
100 200 125
201 210 89
900 1000 174
10 1 20
```

É importante ter muita certeza, porque se submetermos algo errado para o cliente, ele ficará possesso. Por isso desenvolvemos várias estratégias para não errar. Vamos ficando cada vez mais rápidos ao fazer testes e verificações.

Assim, vamos extrapolar esse caso e fazer os demais. Voltando para a `entrada4.txt` :

```
1 10
100 200
201 210
900 1000
10 1
200 100
210 201
1000 900
```

E o mesmo para a `saida4.txt`.

```
1 10 20
100 200 125
201 210 89
900 1000 174
10 1 20
200 100 125
210 201 89
1000 900 174
```

Vamos testar? Primeiro sem o `diff`.

```
Alura-Azul:bin alura$ java Main < entrada4.txt
1 10 20
100 200 125
201 210 89
900 1000 174
10 1 20
200 100 125
210 201 89
1000 900 174
```

Parece tudo ok. E com o `diff`?

```
Alura-Azul:bin alura$ java Main < entrada4.txt | dff - saida4.txt
Alura-Azul:bin alura$ java Main < entrada4.txt
```

Tudo como o esperado. Vamos agora rodar as outras entradas, para ter certeza de que tudo está funcionando direito para elas também. Começaremos pela `entrada1.txt`.

```
Alura-Azul:bin alura$ java Main < entrada1.txt
1 10 20
100 200 125
201 210 89
900 1000 174
```

A seguir, temos a `entrada2.txt`, que ainda não corrigimos.

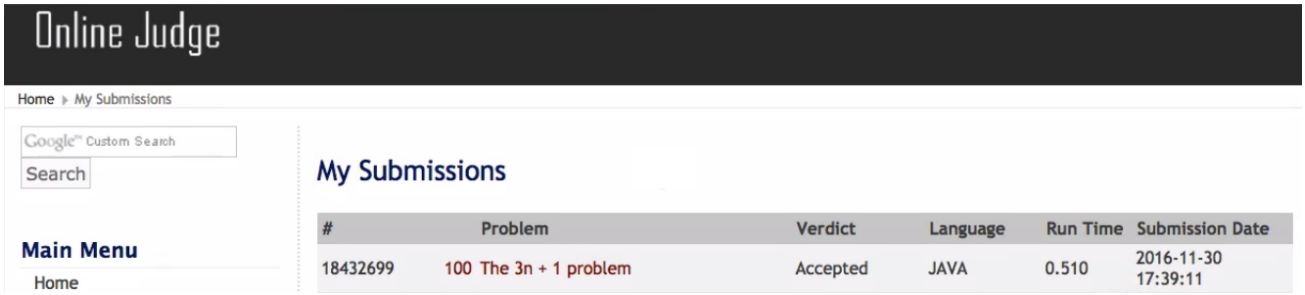
```
Alura-Azul:bin alura$ java Main < entrada2.txt
1 10 20
100 200 125
201 210 89
900 1000 174
```

Continua demorando demais, então interromperemos sua leitura. E a `entrada3.txt` ?

```
Alura-Azul:bin alura$ java Main < entrada3.txt
1 10 20
100 200 125
201 210 89
900 1000 174
```

Está funcionando bem. Vamos submeter novamente para o cliente? Ainda temos um teste nosso que demora demais, e se o cliente fizer o mesmo teste, ele vai perceber o problema. Acabamos de olhar o código com calma e fizemos correção ainda nas primeiras linhas.

Vamos submeter a solução no UVa novamente. Depois devemos ir em `My Submissions` para ver o resultado.



Foi aceita! Demorou meio segundo, mas passou. Mesmo havendo casos que o cliente avisou que poderiam acontecer, e no nosso computador não funcionaram. No computador do cliente, funcionou. O que aprendemos com isso? Nosso cliente explicita algumas coisas que são exageros, e deixa implícitas coisas importantes que temos que deduzir. Tudo isso acontece, pois ele são humanos e nós também. E precisamos saber nos adaptar a isso.

Em uma maratona de programação de verdade, eu corrigiria esse problema antes de submeter. Isso porque não sabemos o que o juiz fará com o código, e se ele rodar esse teste que não dá certo, perderíamos pontos. Não queremos perder ponto à toa em um campeonato internacional de programação, não é?

Nesse nosso caso, o exercício demonstrou para nós a insegurança que temos frente ao cliente. E temos que aprender a conversar com ele para sanar as nossas dúvidas. Às vezes ele pedirá mais que o necessário, às vezes menos, e temos que lidar com isso. Um programador sênior sabe prever essas questões, implementar soluções simples e conversar com o cliente de maneira aberta e direta para que o mínimo possível seja o suficiente. Não é implementar menos do que o necessário, é implantar o mínimo para fazer o que o cliente quer e precisa. No momento em que o nosso cliente, ou juiz, falar que está tudo bom e funcionando, ficamos satisfeitos.

Em breve mostrarei para você que é assim que eu trabalho no meu dia a dia, como programador com mais experiência. Eu programo há 20 anos, e hoje em dia escrevo linha a linha pensando em cada caso de erro possível. Se escrevo o `for()` , me pergunto se o maior número é aquele mesmo, se o `atual++` não vai passar do valor de `32 bit` e o Java não vai aguentar. Ao chamar o `resultado` , será que o valor que definimos como mínimo do ciclo é válido? É nesse tipo de coisa que temos que pensar conforme escrevemos. Claro que por ser um programador sênior e ter mais experiência, consigo pensar rapidamente nessas questões.

Claro, se você tiver acesso a testes de unidade, crie esses testes. Trabalhamos com testes de unidades no dia a dia ao lidar com sistemas maiores, sem entradas e saídas de output tão simples como essas. Na maratona, acabamos por fazer esses testes de `.txt` mesmo, mas para o cotidiano sugiro os cursos de testes de unidade.

O que ainda quero mostrar para vocês é como eu realmente resolvo esses problemas em uma maratona. Você não precisa chegar nesse ritmo imediatamente. Eu participei de maratonas durante 7 anos, treinando lógica e resolução de problemas, treinando pensar adiante, e no que poderia atrapalhar o meu código. Fiquei quase todos os dias nesse período treinando isso. O ritmo vem com o tempo e com a experiência, portanto pratique. O mais importante é que esteja consciente da sua situação, é isso que te deixa à frente de quem está apenas escrevendo. É isso que quero que fique como mensagem para você. A seguir, vou mostrar como é que solucionamos problemas metralhando, para quem quiser acompanhar. Até lá!