Maratona de Programação 101

Introdução a competições de programação de computadores

Antonio Cesar de Barros Munari
Faculdade de Tecnologia de Sorocaba

As competições de programação conseguiram alcançar uma grande popularidade entre os estudantes dos diversos estágios de formação escolar ao redor do mundo. Das competições visando os ensinos fundamental e médio, como a IOI - *International Olympiad in Informatics* (http://ioinformatics.org/index.shtml) até o ensino de graduação, como o ACM ICPC - *International Collegiate Programming Contest* (http://cm.baylor.edu/welcome.icpc), por exemplo, milhares de alunos e escolas são mobilizados anualmente por esses eventos.

A crescente dinamização no Brasil da Maratona de Programação (http://maratona.ime.usp.br/) promovida pela Sociedade Brasileira de Computação (SBC) e que é parte integrante do ACM ICPC ilustra bem essa percepção de popularização: sua última edição, ocorrida no ano de 2011, contou com a participação de um total de 536 equipes representando 191 escolas diferentes. Esse crescimento tem atraído competidores, docentes e escolas dos mais variados perfis e aqueles que participam pela primeira vez rapidamente percebem que pode não ser muito fácil adaptar-se ao formato das provas e familiarizar-se com os fundamentos técnicos básicos requeridos para que resultados satisfatórios sejam atingidos. Devido à maneira como as competições são organizadas e operacionalizadas, alguns aspectos podem diferir significativamente dos ambientes práticos das aulas de programação, tanto nos recursos utilizados como também quanto a critérios de avaliação e o desconhecimento de tais elementos costuma tornar confusa e até mesmo frustrante a preparação e participação desse público iniciante.

Como existe no Centro Paula Souza o interesse pela participação de um maior número de suas Fatecs nessas competições, motivado pelo entendimento de que elas promovem a criatividade, a capacidade de trabalho em equipe, a habilidade de resolver problemas sob pressão e também estimulam o desejo de aprimoramento técnico dos estudantes, torna-se estratégico atenuar para as diversas unidades essa curva de aprendizado requerida ao trabalho de preparação para esse tipo de evento. O objetivo deste material é, portanto, apresentar de maneira tão simples quanto possível o formato da competição e os aspectos técnicos mais relevantes para aqueles que ainda não possuem intimidade com competições no estilo da Maratona de Programação. Espera-se com isso estimular o maior número de estudantes e docentes a envolverem-se no processo, permitindo que melhorem seu aprendizado, desenvolvam-se tecnicamente e, por que não, também divirtam-se competindo. Se este material contribuir para o aumento de interesse pelas competições de programação no Centro Paula Souza ou até mesmo para melhorar o nível de nossos competidores, terá alcançado seus objetivos.

Este documento está organizado em diversas seções. A primeira delas vai apresentar rapidamente o formato de uma competição nos moldes da Maratona de Programação. A segunda vai descrever o ambiente de trabalho encontrado pelas equipes participantes desse tipo de competição. Em seguida serão relatadas algumas observações

consideradas importantes sobre as linguagens geralmente utilizadas e problemas básicos encontrados por competidores que estão iniciando sua participação. Na sequência é tratada a maneira que os programas são executados pelos juízes da competição, que por não ser interativa como em geral os alunos estão acostumados, costuma ser fonte de grande volume de dúvidas e confusões. A seção seguinte vai tratar da forma como os programas são testados e quais os tipos de erro que podem ser encontrados pelos juízes em sua avaliação. Por fim um pequeno estudo de caso vai analisar um problema típico de competição e discutir diversas questões técnicas, procurando reforçar tudo que foi abordado nas seções anteriores e ilustrar exemplos de soluções para o problema em diversas linguagens.

Questões técnicas mais específicas, ligadas a lógica de programação, projeto de algoritmos e estruturas de dados ou ainda à otimização de algoritmos estão fora do escopo deste material.

O formato geral da competição

Uma prova da Maratona de Programação tem 5 horas de duração e é disputada por equipes compostas obrigatoriamente por três elementos que procuram resolver a maior quantidade de problemas no menor tempo possível. Toda equipe possui um responsável, o *coach*, que cuida da preparação e dos aspectos administrativos e operacionais da participação da equipe nas competições, mas não interage com seus competidores durante a realização da prova. A quantidade de problemas varia de 6 a 15 problemas, dependendo da edição do evento. Competições em nível local têm os problemas descritos em português, competições em nível sul-americano ou mundial têm os problemas formulados em inglês. Cada problema de uma prova é identificado por um código, que é uma letra do alfabeto, e um nome.

Os problemas possuem graus variados de dificuldade, desde aqueles mais fáceis até os muito difíceis, e exploram lógica básica, pesquisa e ordenação, otimização, grafos, manipulação de *strings*, geometria e teoria dos números, principalmente. A disposição dos problemas na prova é aleatória quanto ao seu grau de dificuldade, ou seja, o primeiro problema pode ser difícil, o segundo pode ser muito fácil, o terceiro pode ser muito difícil, etc. Apesar dessa diversidade de níveis de dificuldade, todos os problemas possuem o mesmo peso: resolver um problema fácil tem o mesmo valor no resultado final que resolver um problema difícil. É então muito importante analisar a prova para selecionar os problemas mais fáceis e resolvê-los primeiro, deixando para depois os problemas mais complicados. Sempre que uma equipe consegue resolver um problema, ela ganha um balão colorido, que é colocado junto à mesa de trabalho da equipe, como mostra a figura 1. Cada problema corresponde a uma cor de balão, e se dois times possuem balões de mesma cor é por que conseguiram resolver o mesmo problema.

Fig. 1: Equipes na Final Brasileira de Maratona de Programação, em 2010.

O critério de classificação das equipes no placar final é determinado, em primeiro lugar, pela quantidade de problemas resolvidos. Equipes que conseguiram a maior quantidade de balões ficam no topo do *ranking*, equipes que conseguiram uma menor quantidade de balões ficam no final. O segundo critério de classificação é a quantidade de tempo que cada equipe necessitou para resolver os problemas, sendo esse tempo medido em minutos decorridos desde o início da competição. Equipes que resolveram a mesma quantidade de problemas são classificadas em ordem crescente de tempo: quem gastou menos tempo aparece antes de quem gastou mais tempo.

Sempre que uma equipe apresenta aos juízes da competição uma solução para um problema, haverá uma avaliação para verificar se a solução está correta. Caso esteja, a equipe é notificada, recebe o balão correspondente ao problema e os juízes anotam com quantos minutos decorridos de competição aquela solução foi recebida para julgamento. Se, por outro lado, a solução for julgada incorreta por algum motivo, a equipe é informada através de uma mensagem padrão e poderá tentar corrigi-la e apresenta-la novamente, quantas vezes considerar necessário. Para um problema que a equipe conseguiu resolver corretamente, o tempo total correspondente será dado pelo momento (em minutos) em que a submissão correta foi feita mais 20 minutos de acréscimo para cada submissão julgada incorreta que a equipe fez para aquele problema.

Para ilustrar essa forma de cálculo vamos supor que uma equipe submeteu para julgamento uma solução aos 90 minutos de competição e essa solução foi julgada incorreta. A equipe foi notificada e depois de algum tempo, descobriu o seu erro e

conseguiu corrigi-lo, enviando novamente para julgamento aos 115 minutos. Houve nova avaliação e os juízes julgaram a solução ainda incorreta, notificando novamente a equipe. Depois de mais algum tempo, a equipe submeteu uma nova solução para aquele problema, aos 143 minutos de competição e esta foi considerada correta pelos juízes. Esse problema vai ser computado com um tempo de 143+20+20 = 183 minutos, pois além da submissão correta ocorrida aos 143 minutos de competição tivemos anteriormente duas submissões incorretas, que acrescentaram 20 minutos cada uma ao tempo total do problema.

O placar da competição vai então refletir esses critérios de avaliação e para ilustrar a forma como ele é elaborado vamos analisar uma competição fictícia, envolvendo as equipes Time_1, Time_2, Time_3 e Time_4 e seis problemas para serem resolvidos. A figura 2 mostra a sequência de todos os eventos de interesse ocorridos durante a competição, e a figura 3 apresenta o placar final correspondente.

Tempo decorrido	Equipe	Problema	Resultado
12	Time_1	B	Correto
18	Time_4	A	Incorreto
23	Time_2	B	Correto
23	Time_4	A	Correto
45	Time_1	C	Incorreto
89	Time_2	F	Correto
90	Time_2	A	Incorreto
92	Time_1	E	Correto
115	Time_2	A	Incorreto
123	Time_2	D	Incorreto
143	Time_2	A	Correto
145	Time_1	A	Incorreto
181	Time_4	D	Incorreto
210	Time_3	D	Incorreto
230	Time_1	A	Correto
242	Time_4	D	Incorreto

Fig. 2: Sequência de submissões em uma competição fictícia.

#	Equipe	A	B	C	D	E	F	Total	Tempo
1	Time_2	3/143	1/23		1/-		1/89	3	295
2	Time_1	2/230	1/12	1/-		1/92		3	354
3	Time_4	2/23			2/-			1	43
4	Time_3				1/-			0	0

Fig. 3: Placar final da competição fictícia.

Em geral o placar completo de uma prova vai estar organizado de maneira muito semelhante à figura 3. Nas colunas correspondentes aos problemas são indicadas as submissões de cada time. O valor 3/143 encontrado na intersecção da coluna do problema A com a linha referente à equipe Time_2, por exemplo, significa que foram feitas três tentativas para aquele problema, sendo que a primeira submissão correta ocorreu aos 143 minutos de prova. Um valor 2/- como aquele encontrado na intersecção do problema D com a equipe Time_4 significa duas tentativas realizadas para o problema e nenhuma delas foi julgada correta. Observe que são computados no resultado final de cada equipe apenas os problemas que ela conseguiu resolver durante a competição. A coluna 'Total' indica quantos problemas foram resolvidos por cada equipe, e está em ordem decrescente de seus valores. A coluna 'Tempo' é o somatório dos tempos com as penalidades de 20 minutos para as submissões incorretas, e está em ordem crescente para as equipes que possuem o mesmo valor na coluna 'Total'.

Ambiente utilizado por uma equipe durante a competição

Durante a competição uma equipe recebe um texto descrevendo o enunciado dos problemas com alguns exemplos de entradas possíveis e as respectivas respostas corretas e também dispõe de um computador configurado com editor de texto para a codificação dos programas, compiladores para as linguagens permitidas e algum ambiente por meio do qual ela poderá se comunicar com os juízes. O enunciado pode ser disponibilizado de forma impressa ou apenas em formato digital, conforme o porte da competição. Na Maratona de Programação, cada membro da equipe recebe uma cópia impressa dos enunciados. É aconselhável que os times disponham de folhas de papel em branco para a discussão dos problemas e organização das idéias. Durante a prova a consulta é permitida apenas a material previamente impresso em papel: livros, apostilas, listagens de programas, resumos, dicionários, guias de referência rápida, etc. A comunicação entre os membros de um mesmo time é permitida, mas é proibida a comunicação ou troca de material entre as equipes.

O ambiente disponível para o computador pode ser Windows/Dos ou Unix-*like* (alguma distribuição Linux, por exemplo). Para Java costuma existir algum IDE, como Eclipse por exemplo, e para as outras pode estar disponível o Visual C, Visual C++ ou outros IDEs, como DevC++. Um ambiente mínimo e, para muitos, ideal seria um serviço de linha de comando, como a janela `command` do Windows/Dos ou o *prompt* do CygWin (http://www.cygwin.com/), os compiladores básicos `gcc`, `g++` ou `fpc` e um editor de textos como o Notepad++.

Além disso, o computador deve dispor de uma interface para comunicação com os juízes, para que as soluções sejam submetidas, dúvidas sejam relatadas à organização, o placar possa ser consultado, impressões de listagens de programas possam ser solicitadas, etc. Na Maratona de Programação esse ambiente é o Boca (http://www.ime.usp.br/~cassio/boca/), que é acessado por meio de um browser.

Ao abordar um problema, a equipe deve estudar o enunciado atentamente, discutir e identificar o que exatamente deve ser feito pelo programa a ser desenvolvido. Depois, deve criar um conjunto de casos de testes além daqueles existentes no enunciado da prova, de maneira que seja possível explorar as diversas possibilidades e situações limites colocadas pelo problema, além de ajudar aos competidores a melhor entenderem o enunciado. Para cada caso de teste o resultado esperado também deve ser calculado e anotado, para conferência posterior, sendo recomendado que nessa etapa sejam gerados os arquivos de entrada e de saída abordados mais à frente neste material, na seção "Como os programas são executados pelos juízes". Em seguida, deve-se delinear a solução para o problema e codificar, compilar e testar localmente o programa, garantindo que os valores impressos atendem **exatamente** à formatação indicada no enunciado. Quando a equipe entender que o programa está correto e resolve o problema proposto, deverá fazer a submissão dessa solução para os juízes, por meio da interface disponível, que em geral é o Boca. Dentro de alguns minutos os juízes fazem a notificação do resultado, indicando se a solução foi aceita ou se algum tipo de erro foi encontrado. Se o programa foi aceito, a equipe ganha um balão, em caso contrário o time deve procurar corrigir o erro encontrado pelos juízes e submeter novamente a solução.

Considerações sobre a compilação dos programas

As competições de programação podem variar significativamente quanto às linguagens de programação permitidas. Algumas competições e sites de treinamento permitem uma grande variedade delas, como o Google Code Jam (http://code.google.com/codejam/) e o SPOJ (http://br.spoj.pl/), enquanto que outras permitem apenas uma variedade muito pequena, como ocorre no UVa Online Judge (http://uva.onlinejudge.org/), TopCoder (http://www.topcoder.com/) e nas competições do ICPC da ACM, da qual a Maratona de Programação brasileira é parte integrante. Considerando apenas a Maratona de Programação, são permitidas as linguagens C, C++ e Java, sendo que durante muito tempo também foi permitida a linguagem Pascal, que hoje em dia é utilizada em apenas algumas sedes da primeira fase da competição. De forma geral as linguagens mais utilizadas nas competições são C++ e, em segundo lugar, a linguagem C.

Exceto pelo conjunto de linguagens admitidas na competição, que é previamente definido pela organização, a equipe possui total liberdade para escolher em qual linguagem ela vai resolver um problema. Isso significa que é possível a um mesmo time submeter, na mesma competição, a solução para um problema em uma linguagem (Java, por exemplo) e para outro problema uma solução em C.

Há muitos compiladores disponíveis para essas linguagens mais tradicionais, e isso pode ser um problema para alguns competidores, pois existem recursos que são específicos de determinados compiladores, mas não são disponíveis em outros. Apesar

de a organização de cada competição poder, eventualmente, adotar o compilador de sua escolha para gerar os executáveis codificados em cada linguagem, alguns produtos costumam ser predominantes.

O compilador mais usado para linguagem C é o `gcc` (que significa *GNU Compiler Collection*) disponibilizado de forma gratuita (http://gcc.gnu.org/) e encontrado em uma série de ambientes operacionais. Para a linguagem C++ costuma ser utilizado o `g++`, que também é parte da coleção GCC. O compilador Pascal de uso mais comum parece ser o `fpc`, chamado de *Free Pascal* (http://www.freepascal.org/) e que é um produto de código aberto. Para Java costuma-se utilizar implementações Sun JDK e IBM.

Existe uma série de enganos muito comuns cometidos por competidores que recebem erros de compilação por motivos primários, o que é especialmente frustrante. O erro mais primário de todos é submeter uma solução feita em uma linguagem, como o C++ por exemplo, e indicar no ato da submissão que a linguagem utilizada foi outra, como Pascal ou Java. O programa feito em uma linguagem vai então ser compilado como se tivesse sido escrito em outra e, fatalmente, erros de sintaxe serão encontrados. É basicamente um caso de desatenção, que às vezes demora para ser percebido no calor de uma competição.

Outro motivo para erros de compilação é o uso de recursos específicos de um compilador que não são disponíveis no compilador utilizado pelos juízes. Um caso clássico ocorre em C, onde alguns competidores costumam usar recursos da biblioteca `conio.h` para tratamento de I/O via console que, apesar de encontrada em diversos ambientes de desenvolvimento, não é padrão da linguagem e não está disponível no `gcc`. A recomendação básica é que o competidor faça uso apenas de recursos garantidamente suportados pelo compilador utilizado pelos juízes da competição. Um conhecimento dos recursos que são padrão da linguagem e daqueles que são específicos de um ou outro produto em particular é muito importante para qualquer competidor.

Como os programas são executados pelos juízes

O programa compilado com sucesso é executado no ambiente de julgamento por meio de uma instrução que faz o redirecionamento tanto da entrada como da saída. Um exemplo de chamada de programa nessa forma seria:

```
C:\Temp\prog1.exe < arqEntrada.txt > arqSaida.txt
```

Os operadores '<' e '>' utilizados nessa sintaxe implementam, respectivamente, os chamados redirecionamento de entrada e de saída, tanto em ambientes Unix-*like* como Windows/Dos e permitem que o programa execute um conjunto de entradas previamente definidas armazenadas em formato texto no arquivo `arqEntrada.txt` e imprima os resultados do processamento no arquivo `arqSaida.txt`. Obviamente, os nomes desses dois arquivos são definidos pelo usuário.

As linguagens de programação possuem instruções destinadas a receber os dados que serão processados pelo programa e que serão lidos a partir de algum dispositivo *default*. Esse dispositivo é chamado de entrada padrão (*standard input*) e corresponde ao teclado, que é onde instruções como `scanf` (em linguagem C) ou `read` (em Pascal)

buscam os valores a serem utilizados, por padrão. É possível modificar esse comportamento no ato da chamada do programa, de maneira totalmente externa ao código nele contido por meio do operador '<'. Isso significa que a instrução `scanf` ou `read` do programa passará a buscar automaticamente o conteúdo a ser lido no arquivo indicado por esse operador, sem que a lógica do programa precise tratar isso explicitamente. De maneira análoga, o mesmo conceito se aplica a instruções de saída, como `printf` (em C) ou `writeln` (em Pascal), cuja saída *default* é a tela, mas que pode ser redirecionada para um arquivo texto específico por meio do operador '>' na sintaxe de chamada do programa na linha de comando.

Para ilustrar melhor a forma como as entradas são manipuladas, vamos considerar como exemplo um programa que calcula o resultado final dos alunos em uma disciplina. Para isso ele receberá, para cada aluno, a nota de duas provas e a sua quantidade de faltas. Ele calculará a média aritmética das notas do aluno e, se essa média for igual ou superior a 6 e a respectiva quantidade de faltas for inferior a 10 ele deverá imprimir a *string* "Aprovado", e em caso contrário, imprimirá "Reprovado". Vamos assumir ainda que o programa deverá processar os resultados de vários alunos, parando quando forem informados para um aluno três números negativos, ou seja, para as provas e total de faltas.

Ao testar manualmente um programa desses, teríamos que digitar os três valores para o primeiro aluno e esperar o programa imprimir o resultado correspondente, em seguida informar três novos valores, para o segundo aluno e aguardar o resultado para esse novo caso, e assim por diante, até o momento em que informaríamos três valores negativos quaisquer, para a primeira prova, para a segunda prova e para o total de faltas, sinalizando que o programa deveria encerrar sua execução. Vamos imaginar uma possível execução do programa para tratar os resultados de 4 alunos:

1. Usuário inicia a execução do programa.
2. Usuário informa (por meio do teclado) o valor 3.0 para a primeira prova, 6.0 para a segunda prova e 5 para o total de faltas.
3. Programa computa os valores lidos da entrada e imprime (na tela do computador) o resultado "Reprovado".
4. Usuário informa o valor 6.0 para a primeira prova, 7.0 para a segunda prova e 12 para o total de faltas.
5. Programa computa os valores lidos da entrada e imprime o resultado "Reprovado".
6. Usuário informa o valor 6.5 para a primeira prova, 5.5 para a segunda prova e 8 para o total de faltas.
7. Programa computa os valores lidos da entrada e imprime o resultado "Aprovado".
8. Usuário informa o valor 10.0 para a primeira prova, 10.0 para a segunda prova e 0 para o total de faltas.

9. Programa computa os valores lidos da entrada e imprime o resultado "Aprovado".
10. Usuário informa o valor -1 para a primeira prova, -3 para a segunda prova e -5 para o total de faltas.
11. Programa computa os valores lidos e encerra o processamento, sem imprimir nada na tela.

Essa modalidade de execução do programa é chamada de **interativa**, pois a cada intervenção do usuário o programa executa algum processamento, como um diálogo entre homem e máquina. É uma abordagem intuitiva, simples, porém lenta e trabalhosa, que torna muito difícil a depuração e teste de programas. Para testarmos um grande conjunto de valores, correspondentes em nosso exemplo a centenas ou milhares de alunos, dispenderíamos uma quantidade muito grande de tempo e de trabalho com a digitação. Se percebêssemos algum erro no comportamento do programa teríamos que corrigi-lo e testá-lo novamente, inserindo todos os dados utilizados no teste anterior, em um retrabalho desanimador. Em vez disso podemos nos beneficiar da execução com redirecionamento da entrada. Inicialmente criamos com um editor de texto qualquer um arquivo em formato texto com os valores a serem processados. A figura 4 mostra como poderia ser o conteúdo desse arquivo para os dados de teste utilizados no exemplo do nosso programa que verifica quem foi aprovado ou reprovado:

```
3.0 6.0 5
6.0 7.0 12
6.5 5.5 8
10.0 10.0 0
-1 -3 -5
```

Fig. 4: Exemplo de arquivo de entrada para o problema do resultado dos alunos

Observe que apenas os valores efetivamente informados pelo usuário estão ali presentes, na mesma sequência em que seriam digitados em uma execução interativa. Os valores numéricos podem ser separados por espaços em branco, tabulações ou quebras de linha, conforme a preferência do usuário. No caso, os valores referentes a cada aluno foram colocados em uma mesma linha, para facilitar a visualização.

Supondo que tenhamos salvo esse arquivo com o nome `alunos.txt` e que o nome do executável seja `ProgNotas.exe`, ambos contidos na pasta `Temp` do disco `C:`, poderíamos agora testar o programa da seguinte forma:

1. Usuário dispara a execução do programa por meio da instrução

   ```
   C:\Temp\ProgNotas.exe < alunos.txt
   ```

2. O programa lê os três primeiros valores (`3.0 6.0 5`) do arquivo, computa o resultado e imprime na tela a *string* "Reprovado".
3. O programa lê os próximos três valores (`6.0 7.0 12`) do arquivo, computa o resultado e imprime na tela a *string* "Reprovado".

4. O programa lê os próximos três valores (`6.5 5.5 8`) do arquivo, computa o resultado e imprime na tela a *string* "Aprovado".
5. O programa lê os próximos três valores (`10.0 10.0 0`) do arquivo, computa o resultado e imprime na tela a *string* "Aprovado".
6. O programa lê os próximos três valores (`-1 -3 -5`) do arquivo e encerra o processamento sem imprimir nada na tela.

Observe que agora a intervenção do usuário ficou reduzida apenas à chamada do programa, pois todos os dados serão buscados automaticamente no arquivo de entradas, com as respostas sendo impressas na tela à medida que o processamento for ocorrendo. Se precisarmos testar novamente o programa com as mesmas entradas, basta digitar mais uma vez a mesma instrução de execução com o redirecionamento de entrada e o processamento anteriormente descrito se repetirá. Podemos dizer que essa forma não-interativa de execução e teste do programa é um tipo de execução **em lote**, pois todo o conjunto de entradas é processado de uma única vez, não havendo espaço para intervenção do usuário entre o início e o final do processamento. Dessa forma, é importante que o programa não contenha instruções que esperam algum tipo de ação do usuário, como por exemplo a espera de que ele pressione alguma tecla para continuar.

Se também fizermos o redirecionamento da saída, os valores que seriam impressos na tela passarão a ser gravados no arquivo especificado. Assim, caso o usuário disparasse a execução do programa anterior por meio da instrução

```
C:\Temp\ProgNotas.exe < alunos.txt > resultados.txt
```

o arquivo de saída teria, ao final do processamento, o conteúdo indicado na figura 5.

```
Reprovado
Reprovado
Aprovado
Aprovado
```

Fig. 5: Exemplo de arquivo de saída para o problema do resultado dos alunos.

Podemos, portanto, esquematizar a execução do programa com os redirecionamentos de entrada e de saída através da figura 6.

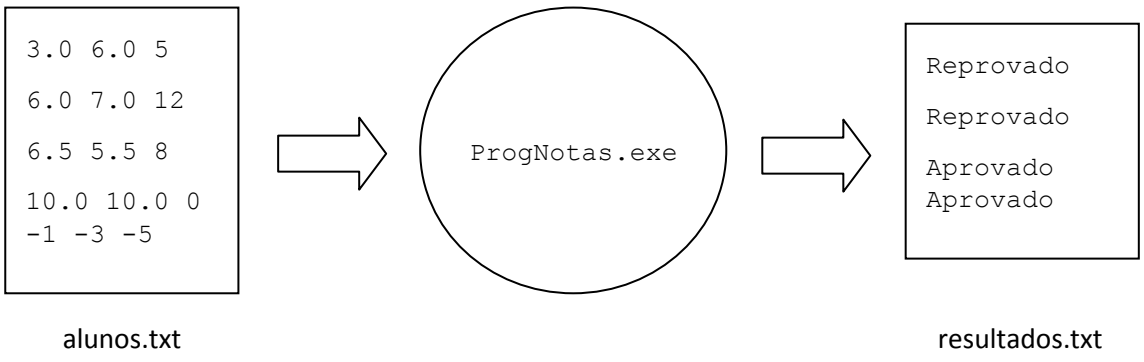

Fig. 6: Visão esquemática da execução de um programa com redirecionamentos de entrada e de saída.

Como os programas são avaliados

Em uma competição de programação o competidor (ou time de competidores) submete para avaliação um código fonte que, acredita-se, pode resolver um determinado problema. Então, a avaliação deve ser rápida e precisa verificar várias coisas:

1. O código fonte é sintaticamente válido, ou seja, ele pode ser compilado? A primeira possibilidade de erro é, portanto, a ocorrência de erros de compilação. Uma submissão que não compila tem o seu julgamento encerrado nesta etapa, reportando uma mensagem do tipo COMPILATION ERROR (CE). Algumas situações típicas que levam a erros de compilação foram abordadas anteriormente neste material.

2. O código fonte apresenta algum tipo de erro de execução? Casos comuns são: estouro de memória, acesso a elementos inexistentes de um vetor (o vetor tem 100 elementos e tentamos acessar o elemento 101, por exemplo), divisão por zero, estouro de pilha (*stack overflow*) com rotinas recursivas, etc. Tais erros de execução geralmente vão ocorrer quando os casos de teste que os juízes utilizam possuem valores críticos, nos limites extremos do domínio do problema, ou em casos especiais implícitos no enunciado. Uma situação muito comum com competidores novatos que programam nas linguagens C e C++ é o programa encerrar sem retornar um código zero para o sistema operacional. Nessas linguagens o ponto de entrada do programa é uma rotina do tipo `int` chamada `main()`. A convenção de chamada para muitos sistemas operacionais, como aqueles Unix-*like*, é que essa rotina principal encerre retornando um código numérico informando se a execução foi bem sucedida ou não. Dessa forma o retorno de um código 0 indica que a execução ocorreu sem erros, qualquer coisa diferente disso significa um término anormal do programa. Ambientes da família Windows são mais liberais (ou permissivos, segundo alguns críticos) e permitem que a rotina `main()` seja do tipo `void`, por exemplo. A figura 7 apresenta dois programas equivalentes, que corretamente recebem dois inteiros e imprimem a sua soma: o programa 7(a) não atende à convenção de chamada e sua submissão geraria um erro de execução por parte dos juízes, enquanto que a versão 7(b) seria considerada correta.

```
#include <stdio.h>
void main( )
{   int v1, v2;

    scanf( "%d %d", &v1, &v2 );
    printf( "%d\n", v1+v2 );
}
```

(a)

```
#include <stdio.h>
int main( )
{   int v1, v2;

    scanf( "%d %d", &v1, &v2 );
    printf( "%d\n", v1+v2 );
    return 0
}
```

(b)

Fig. 7: (a) Programa que gera erro de execução. (b) Programa correto.

Submissões com qualquer tipo de erro de execução são encerradas reportando uma mensagem do tipo RUNTIME ERROR (RE).

3. O programa soluciona o problema com eficiência? Um problema tipicamente possui vários algoritmos distintos capazes de resolvê-lo, com diferenças entre a quantidade de memória requerida e de tempo de execução consumido por cada um. Em competições de programação todo problema possui um limite de tempo máximo dentro do qual uma solução deve resolver todos os casos de teste utilizados pelos juízes. Se um competidor submete uma solução com um algoritmo correto (capaz de resolver corretamente todos os casos de teste propostos pelos juízes) mas que para isso requer mais tempo do que o limite definido para o problema, a solução não será aceita, retornando uma mensagem do tipo TIME LIMIT EXCEEDED (TLE). Algumas situações que levam a esse tipo de erro são programas que ficam em *looping* infinito, outras são algoritmos muito ingênuos e lentos e também ocorrem com equipes novatas que colocam instruções solicitando alguma intervenção do usuário, do tipo "Pressione alguma tecla para continuar", ignorando que a execução do programa é feita em lote e não de forma interativa. Também é importante ter em mente que o limite de tempo é definido para o problema, independentemente de qual a linguagem que será utilizada para codificar a solução. Isso significa que se o limite de tempo de um problema é, por exemplo, 3 segundos, um programa só será aceito se resolver todos os casos de teste dentro desse tempo, não importando se o programa foi feito em C ou em Java. Como Java possui algum tipo de *overhead* de execução (carga da máquina virtual, recursos de entrada de dados mais lentos, altos níveis de abstração, etc), é comum soluções nessa linguagem estourarem o limite de tempo com facilidade para determinados problemas.
4. O programa resolve corretamente o problema? É claro que o programa deve conseguir responder acertadamente a todos os casos de teste possíveis para o problema. É necessário então verificar se as respostas fornecidas pelo programa são iguais às respostas esperadas. A forma como isso é verificado com rapidez nas competições é muito simples: os juízes dispõem de um arquivo de entradas para cada problema e possuem um arquivo contendo as saídas esperadas para o conjunto de casos contido no arquivo de entradas. Esse arquivo de saídas é chamado de gabarito, e foi gerado por um programa que sabidamente resolve corretamente o problema. Quando uma submissão é recebida, os juízes compilam a solução enviada pelo competidor e depois a executam redirecionando a entrada para que leia os dados do arquivo de entradas anteriormente mencionado e redirecionam a saída para um arquivo texto qualquer. Em seguida, fazem a comparação byte a byte entre o gabarito e o arquivo de saídas gerado pelo programa que está sendo testado. Se nenhuma diferença for encontrada, o programa é considerado correto e uma mensagem do tipo ACCEPTED (AC) é emitida, se algum valor produzido pelo programa é diferente do existente no gabarito, o programa é considerado incorreto e uma mensagem do tipo WRONG ANSWER (WA) é reportada. Se os dois arquivos de saída forem diferentes apenas quanto à quantidade de espaços em branco ou quebras de linha utilizadas, o programa é considerado incorreto e uma mensagem do tipo PRESENTATION ERROR (PE) é retornada. A comparação entre os dois arquivos é realizada tipicamente pelo comando do sistema

operacional `fc` em ambientes Windows/Dos e pelo comando `diff` nos sistemas Unix-*like*.

Considerando então que ao serem avaliados os programas são executados fazendo tanto o redirecionamento de entrada como de saída, ou seja, o processamento é feito em lote, de forma não interativa, e que tudo aquilo que é impresso pelo programa será colocado em um arquivo que será comparado com o gabarito dos juízes, devemos atentar para o fato que se colocarmos no código mensagens solicitando ao usuário para que faça alguma coisa, essas mensagens serão também colocadas no arquivo de saída. A título de exemplo, considere uma variação do programa da figura 7 mostrado anteriormente que recebe dois números e imprime a soma, encerrando o processamento quando forem recebidos dois números zero.

```
#include <stdio.h>
void main( )
{  int v1, v2;

   while( 1 )
   {   printf( "Informe um numero:\n" );
       scanf( "%d", &v1 );
       printf( "Informe outro numero:\n" );
       scanf( "%d", &v2 );

       if( v1 == 0 && v2 == 0 )
          break;
       else
          printf( "Soma: %d\n", v1+v2 );
   }
   return 0;
}
```

Fig. 8: Programa para a soma de pares de inteiros, interativo.

Suponha agora que os juízes vão testar o programa com cinco pares de números, e que para isso foi criado o arquivo de entradas mostrado na figura 9.

```
5 6
3 0
1 2
9 29
0 0
```

Fig. 9: Possível arquivo de entrada para o problema da soma de pares de números.

Nesse caso, o arquivo de saídas dos juízes (o gabarito) deverá conter apenas os resultados para as quatro somas indicados no arquivo de entradas, como mostra a figura 10.

```
11
3
3
38
```

Fig. 10: Arquivo de saídas esperadas para o problema da soma de pares de inteiros.

Agora, vamos tentar executar o programa apresentado na figura 8 utilizando como entrada o arquivo descrito na figura 9 e redirecionando a sua saída para um arquivo texto qualquer, como seria feito pelos juízes. O conteúdo desse arquivo de saída seria o apresentado na figura 11.

```
Informe um numero:
Informe outro numero:
Soma: 11
Informe um numero:
Informe outro numero:
Soma: 3
Informe um numero:
Informe outro numero:
Soma: 3
Informe um numero:
Informe outro numero:
Soma: 38
Informe um numero:
Informe outro numero:
```

Fig. 11: Arquivo com as saídas produzidas pelo programa da figura 8.

Observe que os valores numéricos correspondentes às somas estão corretos no arquivo, mas junto a uma série de outros conteúdos, que são as mensagens emitidas para o usuário. É claro que ao compararmos os arquivos das figuras 10 e 11, eles não serão considerados iguais, apesar de o programa, tecnicamente, estar correto. Então precisamos ter em mente que em competições de programação os programas só devem imprimir os resultados esperados, e não mensagens de diálogo com o usuário. Assim, uma versão do programa da figura 8 que produziria o resultado correto, no formato do arquivo de saída descrito na figura 10, seria aquele apresentado na figura 12.

```
#include <stdio.h>
void main( )
{  int v1, v2;

   while( 1 )
   {   scanf( "%d", &v1 );
       scanf( "%d", &v2 );

       if( v1 == 0 && v2 == 0 )
          break;
       else
          printf( "%d\n", v1+v2 );
   }
   return 0;
}
```

Fig. 12: Programa para a soma de pares de inteiros, em lote.

Devemos entender que as questões ligadas aos diálogos são secundárias do ponto de vista dos algoritmos de programação, e são melhor colocadas em disciplinas específicas de Interface Homem-Máquina ou correlatas. Competições de programação tratam apenas da lógica fundamental que resolve um problema: tendo recebido de alguma forma todos os dados necessários, o programa consegue atingir o resultado esperado?

Finalizando esta seção que trata da forma de julgamento das submissões em competições de programação, apresentamos na figura 13 um fluxograma que representa as linhas gerais do procedimento de verificação de uma submissão.

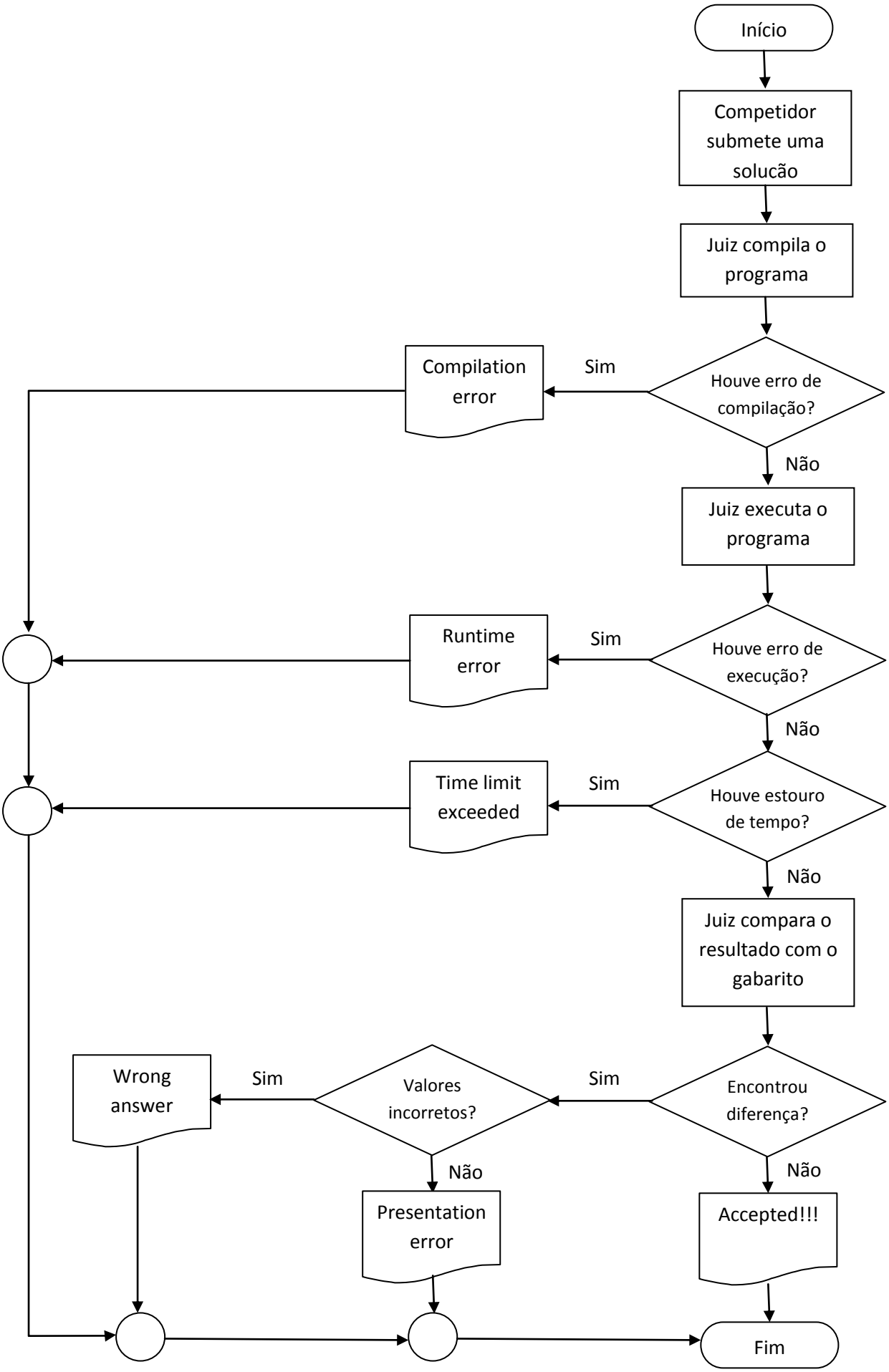

Fig. 13: Fluxograma do julgamento de uma submissão.

Estudo de caso: Portas

Vamos agora proceder a um rápido estudo de caso visando consolidar tudo o que foi apresentado até o momento. Analisaremos um problema utilizado na primeira fase do primeiro InterFatecs de Programação.

O enunciado

Problema A

Portas

Nome do arquivo fonte: `portas.c`, `portas.java`, `portas.cpp` *ou* `portas.pas`

Autor: Leandro Luque (Fatec Mogi das Cruzes)

Durante as férias, João Pedro gosta de aproveitar o tempo brincando com os amigos do prédio onde mora. Uma de suas brincadeiras preferidas é o "Peão Abre-Fecha Porta". A brincadeira começa com a escolha de um dos participantes, o peão. Em seguida, os outros participantes definem um número e o peão deve passar por todos os andares do prédio que sejam múltiplos desse número para inverter o estado da porta do apartamento localizado no andar – ou seja, se a porta estiver aberta, ele a fechará; se estiver fechada, ele a abrirá. As portas estão inicialmente todas fechadas e existe apenas um apartamento por andar. A brincadeira segue com os participantes definindo novos números e o peão abrindo/fechando as portas dos andares múltiplos desses números. Quando o grupo desejar, o peão estiver muito cansado, ou algum morador reclamar, cada participante da brincadeira, com exceção do peão, é questionado sobre o estado das portas do prédio em cada andar (na ordem do mais baixo para o mais alto). Aquele que acerta, ganha um doce e fica livre de ser peão durante todo o dia.

Como João Pedro adora doce e é um pouco preguiçoso, há tempos ele vem procurando por alguma forma de sempre ganhar a brincadeira. Para isso, ele pediu para o seu pai, um especialista em Informática, para desenvolver um programa que, dados os números que serão especificados pelos participantes, determina o estado final das portas dos andares.

Entrada

A entrada do programa é composta por diversos casos de teste. A primeira linha de cada caso de teste contém dois números inteiros *A* e *N*, separados por espaço, indicando o número de andares (variando de 1 a 100) e a quantidade de números que serão informados pelos participantes (variando de 1 a 200), respectivamente. Cada uma das *N* linhas seguintes contém um dos números especificados pelos participantes (variando de 1 a *A*). O último caso de teste é seguido por uma linha que contém dois zeros separados por um espaço em branco.

Saída

Para cada caso de teste, imprima uma linha contendo *A* caracteres, indicando o estado de cada uma das portas dos andares do prédio (o caractere mais à esquerda representa o andar mais baixo; o mais à direita representa o andar mais alto). Caso a porta do andar esteja aberta, imprima o caractere 'O'. Caso a porta esteja fechada, imprima o caractere 'C'.

Exemplo de entrada	Exemplo de saída
10 5 2 4 9 10 1 5 3 3 4 2 0 0	OCOOOCOOCO COOCC

Análise geral do enunciado

O problema apresentado foi o primeiro do caderno de enunciados, e por esse motivo foi identificado como 'Problema A'. Em uma prova com 9 problemas, esse problema de lógica básica (ou *ad-hoc*) foi o terceiro em quantidade de soluções corretas, tendo sido resolvido com sucesso por 5 das 23 equipes participantes. O nome do problema 'Portas' serve apenas para facilitar a comunicação entre os envolvidos na competição. Como a competição permitiu respostas nas linguagens C, C++, Pascal e Java, o enunciado explicita os nomes que os arquivos-fonte devem receber em cada uma delas. Em seguida ao nome do *problem setter* (a pessoa responsável pela elaboração do problema) temos quatro partes no enunciado: uma contextualização, que descreve uma situação caracterizando o problema; a seção 'Entradas', que descreve a estrutura do arquivo de entradas que o programa deverá processar; a seção 'Saídas', que detalha como os resultados devem ser apresentados no arquivo de saídas e, por fim, os exemplos de entradas e saídas para ilustrar melhor a natureza do problema. Devemos ter em mente que os juízes testarão as soluções enviadas pelos competidores com vários outros casos de teste além dos contidos nesses exemplos. Para ilustrar esse aspecto convém informar que o arquivo de entradas deste problema 'Portas' que foi usado pelos juízes durante a competição tinha cerca de 271 Kb de tamanho, enquanto que os exemplos de entradas contidos no enunciado mal chegavam a 1 Kb.

O texto introdutório que descreve o problema propriamente dito pode ser extenso ou muito breve, mas espera-se que seja auto-explicativo. Dúvidas no seu entendimento devem ser resolvidas por meio da releitura atenta do enunciado, análise dos exemplos de entrada e saída, discussão com os demais membros da equipe e, caso ainda assim restem aspectos importantes não suficientemente compreendidos, pode-se fazer questionamentos por escrito por meio da opção 'Clarifications' do ambiente Boca.

A seção 'Entradas' informa como estarão os dados no arquivo de entradas. Em nosso exemplo sabemos que encontraremos inicialmente dois valores inteiros, que o enunciado chama de A e N para facilitar a descrição que vem a seguir. Devemos

entender que podemos ter em nosso programa variáveis para esses valores com esses mesmos nomes (*A* e *N*) ou podemos também utilizar quaisquer outros nomes para essas variáveis. É informado a seguir que os valores do inteiro *A* estarão na faixa de 0 a 100 e que os valores do inteiro *N* estarão compreendidos entre 0 e 200. Essa informação é muito importante (em alguns casos é absolutamente crítica), mas costuma gerar muita confusão por parte de competidores com pouca experiência. Como devemos interpretá-la? Qual a consequência prática dela para o nosso código? O competidor deve entender que os valores que o programa receberá estarão **sempre** nos intervalos informados. No caso, **nunca** será lido um valor de *A* menor que zero ou superior a 100. Não precisaremos nem mesmo validar esse valor para garantir que ele se encontra na faixa de 0 a 100 por que essa garantia já está sendo dada naquele ponto do enunciado. O mesmo vale para o inteiro *N*: nosso programa nunca encontrará um valor fora da faixa de 0 a 200. É importante enfatizar esse ponto por que é comum encontrarmos competidores gastando um tempo precioso tentando validar a entrada desnecessariamente. Mas, se não precisamos validar, para que serve então essa informação sobre as faixas de valores que serão lidos pelo programa? Basicamente para podermos determinar o tipo de dados das variáveis de entrada e para estimarmos as faixas de valores que o programa vai gerar. Como *A* é um inteiro que sempre estará entre 0 e 100, ele pode ser representado em nosso programa por meio de uma variável inteira de um byte com sinal (tipo `char` em C ou `ShortInt` em Pascal, por exemplo). Já o inteiro *N* requer pelo menos uma variável inteira de um byte sem sinal (tipo `unsigned char` em C ou `Byte` em Pascal, por exemplo). Se pretendemos representar em nosso programa o prédio como um *array*, ele precisará ter pelo menos 100 elementos. Conforme as características do problema, as faixas de valores podem dizer, implicitamente, que um algoritmo mais eficiente pode ser necessário para não haver estouro de tempo, por exemplo.

A seção 'Saídas', por sua vez, detalha exatamente como deve ser apresentado o resultado da computação. E sabemos que o programa deve ser extremamente preciso com relação à formatação da saída, sob o risco de um algoritmo correto receber vereditos Wrong Answer ou Presentation Error dos juízes apenas por pequenas desatenções na impressão das respostas. No problema 'Portas' espera-se que para cada caso de teste, seja impresso exatamente um caracter para cada andar, e esse caracter pode ser apenas uma letra 'O' (maiúscula) ou 'C' (também maiúscula). Se imprimirmos as letras certas, mas em minúsculas, a saída estaria incorreta. Se imprimirmos as letras certas, em maiúsculas, mas com espaços em branco entre elas, a saída estará incorreta. Se houver uma linha em branco entre a resposta de um caso e outro, também um erro será reportado. Espera-se, então, que o programador seja meticuloso, minucioso, detalhista ao imprimir os resultados. Caso haja dúvidas sobre a correta formatação da saída que não podem ser respondidas com base no enunciado, a equipe pode fazer um questionamento aos juízes por meio da já mencionada opção "Clarifications" do Boca.

Como podemos perceber, uma leitura atenta do enunciado é um requisito básico para uma equipe ser bem sucedida em uma competição. Quanto maior a experiência do competidor, maior a facilidade e a rapidez para compreender o enunciado e fazer bom uso das informações que ele contém.

Estratégia para resolver o problema

A forma como uma equipe aborda um problema e o resolve durante uma competição é muito particular, pois depende do nível de conhecimento técnico dos seus membros, da quantidade de experiência acumulada em competições, do nível de dificuldade do problema, das características da linguagem utilizada, das circunstâncias da competição, etc. Assim, não seria muito realista de nossa parte preconizar uma abordagem 'certa' em oposição a todas as outras abordagens 'erradas' possíveis. Afinal temos que reconhecer que há muito espaço para individualização na ação das equipes e, no final das contas, por estarmos falando de competição, a medida de ouro do sucesso ou fracasso, pelo menos durante a prova, é se a abordagem adotada funcionou ou não para a equipe. Apesar disso, podemos sugerir uma abordagem básica que parece ser útil para equipes iniciantes e mesmo para equipes já com alguma experiência. Cabe aos membros da equipe, juntamente com o respectivo *coach*, discutir e experimentar variações a partir dela que se ajustem mais ao perfil dos competidores.

Como vimos, a questão fundamental relacionada ao sucesso ou fracasso na tentativa de resolução de um problema é o correto entendimento do enunciado. Não podemos ser produtivos se não compreendemos exatamente o que se espera da solução a ser desenvolvida e quais as possibilidades e casos críticos que nosso programa deverá tratar. Ler atenciosamente o enunciado é fundamental, e as dúvidas devem ser anotadas e discutidas até haver consenso entre os membros da equipe sobre o que está sendo solicitado no problema. Ás vezes um problema apresenta-se como aparentemente fácil, porém, ao prestarmos mais atenção aos detalhes, verificamos que essa aparência é totalmente enganadora a ponto de decidirmos deixa-lo de lado inicialmente em favor de algum outro problema mais simples. É importante analisar calmamente os exemplos de entrada e saída, e tentar reproduzir à mão os cálculos que solucionaram cada caso de teste apresentado nesses exemplos. Muitos detalhes e particularidades do problema podem ser percebidos nessa tentativa de reprodução manual do processamento sugerido nos exemplos.

Como a finalidade desses exemplos é apenas reforçar o entendimento do enunciado, a equipe deve também procurar imaginar outros casos de teste válidos (e **apenas** casos válidos segundo o que especifica o enunciado), explorando situações críticas (maior valor possível de uma entrada, menor valor, etc) e determinar os resultados esperados para cada teste. No problema 'Portas', com certeza devemos gerar casos de teste com o menor valor útil possível para A (no caso, esse valor é 1) e também com o maior valor possível (que é 100). Também devemos testar com vários valores de N, mesmo para quando o valor de A for 1, pois será útil para melhor compreender particularidades do problema. Também é bom gerar um teste com um alto valor tanto para A como para N, tendo o cuidado de especificar entre os vários inteiros que estarão entre os N valores seguintes números pequenos, para possibilitar aferir se o programa é rápido ou não. Massas de teste que demandam muito processamento podem ser inviáveis de produzir e, principalmente, de terem seu resultado calculado a mão durante a prova, mas mesmo assim devemos tentar gerar pelo menos um caso mais trabalhoso, para verificar se não ocorrerá nenhum erro de execução ou um estouro do limite de tempo. Como a ideia ao testar um caso mais volumoso não é verificar se a resposta dada pelo programa está

certa ou errada, pois isso seria inviável, mas sim observar o comportamento geral do programa na presença de um entrada tão grande, não devemos tentar calcular à mão o valor esperado, e apenas aceitamos a saída dada como correta, caso o programa encerre sua execução normalmente. Para esses casos de teste maiores podemos gerar os dados com o famoso copiar e colar ou então por algum programa simples criado pela equipe especialmente para essa finalidade.

Em seguida, um membro do time deve criar no editor de textos o arquivo de entradas com os exemplos do enunciado e os outros casos elaborados pela equipe e também criar o respectivo arquivo de saídas, com as respostas esperadas dos casos em que é viável determinar a resposta correta. Um bom exemplo de arquivo de entradas para esse problema é apresentado na figura 14, onde os dois primeiros casos de teste foram copiados do enunciado e os restantes foram criados com base nos argumentos discutidos anteriormente. Para esse conjunto de testes o arquivo de saídas esperado é apresentado na figura 15.

```
10 5
2 4 9 10 1
5 3
3 4 2
1 1
1
1 5
1 1 1 1 1
2 5
1 2 1 2 1
100 20
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71
100 200
 2  4  6  8 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38 40
42 44 46 48 50 52 54 56 58 60 62 64 66 68 70 72 74 76 78 80
82 84 86 88 90 92 94 96 98 100
 1  3  5  7  9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39
41 43 45 47 49 51 53 55 57 59 61 63 65 67 69 71 73 75 77 79
81 83 85 87 89 91 93 95 97 99
 2  4  6  8 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38 40
42 44 46 48 50 52 54 56 58 60 62 64 66 68 70 72 74 76 78 80
82 84 86 88 90 92 94 96 98 100
 1  3  5  7  9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39
41 43 45 47 49 51 53 55 57 59 61 63 65 67 69 71 73 75 77 79
81 83 85 87 89 91 93 95 97 99
20 30
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
0 0
```

Fig. 14: Exemplo de arquivo de entradas para testar a solução do problema ‘Portas’.

```
OCOOOCOOCO
COOCC
O
C
OO
COOOOCOOOCOCOCCOOCOCCCOCOCOCOOOOCCCCOCCCOOOCCCOCOCCCCOCCCCCOOOCCO
COOCCOOCCCCCCOCCOCCOCCCCCOCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCC
```

Fig. 15: Saídas esperadas para o arquivo de entradas da figura 14.

Se persistirem dúvidas sobre algum aspecto do enunciado, recomenda-se que a equipe faça uso do recurso 'Clarifications' do Boca. Uma *clarification*, como o próprio termo já diz, é uma solicitação de esclarecimentos direcionada aos juízes da competição. A equipe deve formular o questionamento de maneira clara e específica sobre algum aspecto relevante do problema que foi omitido no enunciado. Não se deve gerar solicitação de esclarecimento apenas para confirmar informações que já estão contidas no enunciado nem pra pedir 'dicas' sobre novos casos de teste ou pra reclamar do compilador utilizado na competição, por exemplo. Em geral os juízes respondem aos pedidos de esclarecimento de forma que todas as equipes possam ver tanto a pergunta como a resposta. Então, se o problema realmente estiver suscitando dúvidas, talvez outra equipe já tenha formulado uma questão correlata aos juízes e a resposta possivelmente estará disponível para consulta na tela de 'Clarifications' do sistema. Se ainda não estiver, é o caso de alguém da equipe solicitar o esclarecimento e aguardar a manifestação dos juízes. No problema 'Portas' praticamente nenhuma questão importante deixou de ser tratada no enunciado, mas poderiam eventualmente ser consideradas pertinentes *clarifications* como:

1. É necessário imprimir alguma resposta para o último caso, em que *A* é zero e *N* é zero, antes de encerrar o processamento? Em caso afirmativo, que resposta seria essa?
2. Devemos deixar uma linha em branco após o resultado do último caso de teste?

Uma vez resolvidas todas as dúvidas sobre o enunciado, é a hora de gerar o programa que resolve o problema, e devemos determinar a estratégia de representação do problema (a sua estrutura de dados) e a lógica correspondente. Um problema como o 'Portas' pode ser resolvido usando um *array* booleano para cada andar (verdadeiro = aberto, falso = fechado), ou um *array* simbólico (armazena a letra 'O' se estiver aberto e 'F' se estiver fechado), usando lógica *bit* a *bit* ou, mesmo, por meio de algum tratamento de *strings*, por exemplo. Se a lógica requerida for mais específica, talvez seja interessante rascunhar em papel a idéia geral do algoritmo ou, conforme o caso, detalhar apenas a estrutura de dados básica e a forma de cálculo de algum valor mais complexo dentro da lógica. Um membro do time deve assumir o computador e codificar o programa, de forma rápida, porém atenciosa, enquanto seus colegas passam a tratar de outros problemas da prova. Finalizada a codificação, deve-se compilar o programa e

testa-lo com o arquivo de entradas produzido anteriormente, direcionando a saída para um arquivo temporário qualquer. Em seguida precisamos comparar o arquivo gerado pelo programa com o arquivo de saídas esperadas criado inicialmente e analisar as eventuais divergências, corrigindo o programa na medida do necessário.

Quando houver consenso entre os membros da equipe de que o problema está correto, fazer a submissão. Se o veredito dos juízes não for favorável, discutir o problema e analisar o código novamente. Se o time não estiver encontrando o erro, uma boa sugestão é solicitar por meio do Boca a impressão de uma listagem do programa, para que o computador possa ser utilizado para codificação e teste de outros programas que já estejam amadurecidos enquanto algum membro da equipe analisa a versão impressa em papel do código fonte do programa com erro.

Exemplos de soluções em C, Pascal, Java e C++

Finalmente, para encerrar a discussão sobre o problema 'Portas' apresentamos três programas que o resolvem corretamente, codificados nas linguagens C, Pascal, Java e C++ e que podem ser úteis como referência para os interessados.

```
/*  Autor: Cesar
*/
#include <stdio.h>

#define MAX_TAM 100
#define TRUE  1
#define FALSE 0

int main()
{     char predio[MAX_TAM];
      int c, A, N, x, numero;

      while( TRUE )
      {     scanf( "%d %d", &A, &N );
            if( A == 0 && N == 0 )
                  break;

            for( c=0; c<A; c++ )
                  prédio[c] = FALSE;

            for( c=0; c<N; c++ )
            {     scanf( "%d", &numero );
                  for( x=numero-1; x<A; x+=numero )
                        predio[x] = !predio[x];
            }

            for( c=0; c<A; c++ )
                  printf( "%c", predio[c] ? 'O' : 'C' );
            printf( "\n" );
      }
      return 0;
}
```

Fig. 16: Uma solução em C para o problema 'Portas'.

```
{  Autor: Cesar
}
Program portas ;
Var predio: Array [ 1..100 ] Of Boolean;
      c, A, N, x, numero: Integer;

Begin
      Read( A );
      Read( N );

      While (A <> 0) And (N <> 0) Do
      Begin
            For c := 1 To A Do
                  predio[c] := False;

            For c := 1 To N Do
            Begin
                  Read( numero );
                  x := numero;
                  While x <= A Do
                  Begin
                        predio[x] := Not predio[x];
                        x := x + numero;
                  End;
            End;

            For c := 1 To A Do
                  If predio[c] Then
                        Write( 'O' )
                  Else
                        Write( 'C' );
            WriteLn;

            Read( A );
            Read( N );
      End;
End.
```

Fig. 17: Uma solução em Pascal para o problema ‘Portas’.

```
// Autor: Abner

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.Arrays;

public class Portas
{  public static void main(String[] args) throws IOException
   { int a, n;
     boolean[] portas = new boolean[100];
     BufferedReader r = new BufferedReader(new
InputStreamReader(System.in));

     String[] l = r.readLine().split(" ");
     a = Integer.parseInt(l[0]);
     n = Integer.parseInt(l[1]);
     while (a + n > 0)
     {  Arrays.fill(portas, 0, a, false);
        while (n-- > 0)
        {  int num = Integer.parseInt(r.readLine());
           for (int i = 0; i < a; i++)
              if ((i + 1) % num == 0)
                 portas[i] = !portas[i];
        }
        String resp = "";
        for (int i = 0; i < a; i++)
           if (portas[i])
              resp += "O";
           else
              resp += "C";
        System.out.println(resp);
        l = r.readLine().split(" ");
        a = Integer.parseInt(l[0]);
        n = Integer.parseInt(l[1]);
     }
   }
}
```

Fig. 18: Uma solução em Java para o problema 'Portas'.

```
// Autor: Celso

#include <iostream>
#include <cstring>

using namespace std;

int main()
{  register int i;
   int a, n, m;
   char portas[101];

   cin >> a >> n;
   while ( a != 0 && n != 0 )
   {  memset( portas, 0, a + 1);

      while ( n-- )
      {  cin >> m;

         for ( i = m; i <= a; i += m)
            portas[i] = !portas[i];
      }

      for ( i = 1; i <= a; i++ )
         cout << (portas[i] ? 'O' : 'C');
      cout << endl;

      cin >> a >> n;
   }

   return 0;
}
```

Fig. 19: Uma solução em C++ para o problema ‘Portas’.