TRABALHO PRÁTICO 1

Programação de Computadores
1º semestre de 2017

O jogo de Nim

Objetivo

O objetivo deste 1º TP, a ser feito individualmente, é programar em Python, o jogo de Nim, como explicado abaixo, para que uma pessoa possa jogar contra o programa.

Proposta

¡ O jogo de Nim, também chamado de jogo de Marienbad, é jogado com 16 palitos, dispostos em 4 linhas, na seguinte situação inicial:

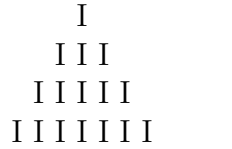

Figura 1: Posição inicial do jogo de Nim

O jogo consiste em retirar quantos palitos o jogador quiser, de uma mesma linha, cada jogador na sua vez. O jogador que retirar o último palito, perde.
O conjunto de `n1` palitos na 1ºa linha, `n2` palitos na 2ºa linha, n3 palitos na 3ºa linha 3 e `n4` palitos na 4ºa linha, pode ser representado pela quadrupla `(n1,n2,n3,n4)`. A situação de início, por exemplo, é representada por `(1,3,5,7)`. O jogador que atingir a situação `(0,0,0,0)` perdeu.
Sejam dois jogadores X e Y. Uma situação `(n1,n2,n3,n4)`, atingida por X, é dita ganhadora se, qualquer que seja a ação de Y, existe uma ação de X que leva Y à situação `(0,0,0,0)`. Quando um jogador X atinge uma situação ganhadora, ele pode evoluir sistematicamente para a vitória, reagindo a cada ação de seu adversário Y com uma ação que o leva de novo a uma situação ganhadora. Esta estratégia permite ao jogador X deixar o jogador Y em uma das situações seguintes: `(1,0,0,0)` , `(0,1,0,0)`, `(0,0,1,0)`, `(0,0,0,1)` e de ganhar a partida. O reconhecimento destas situações ganhadoras é, portanto, de importância crucial.
De fato, é possível determinar, no caso do jogo de Nim, se uma situação é ganhadora ou não, da seguinte maneira:

1. Seja `a1a2a3` o equivalente binário de `n1`, `b1b2b3` o equivalente binário de `n2`, `c1c2c3` o equivalente binário de `n3` e `d1d2d3` o equivalente binário de `n4`.

2. Seja

```
r1 = a1 + b1 + c1 + d1,
r2 = a2 + b2 + c2 + d2,
r3 = a3 + b3 + c3 + d3,
r4 = a4 + b4 + c4 + d4.
```

3. Se `r1, r2, r3,` e `r4` são todos pares, então a situação é perdedora. Senão, é ganhadora.

Por exemplo, a situação inicial `(1,3,5,7)` é uma situação perdedora: em binário,

```
1 é a1a2a3 = 001
3 é b1b2b3 = 011
5 é c1c2c3 = 101
7 é d1d2d3 = 111
```

e `r1 = 2, r2 = 2` e `r3 = 4`, todos pares! Da mesma forma, a situação `(0,0,0,0)` é perdedora, considerando 0 como par. A situação `(1,5,5,0)` é ganhadora: em binário,

```
1 é a1a2a3 = 001
5 é b1b2b3 = 101
5 é c1c2c3 = 101
```

e `r1 = 2, r2 = 0, r3 = 3` e `r4 = 0`.

Programa

O programa Python que você vai entregar deve, ao ser executado, mostrar na tela a situação inicial do jogo de Nim, como mostrado na Figura 1 acima e o escore atualizado de partidas jogadas na atual execução do seu programa. Abaixo da situação de jogo, mostre uma mensagem *curta* que indique qual entrada o programa espera e um pequeno exemplo de jogada.

A jogada será indicada por dois inteiros positivos: a linha da qual se quer tirar palitos, seguido de um inteiro indicando quantos palitos serão retirados dessa linha. Seu programa, então, fará as verificações necessárias (a linha indicada existe? O número de palitos é maior que 0? É possível retirar a quantidade solicitada?), dará mensagens explicando porque não é possível executar o pedido do usuário, se for o caso, e atualizará a situação de jogo após a jogada efetuada.

A apresentação do jogo é importante. Então, lembre-se de limpar a tela após dar as mensagens de erro e atualizar a situação de jogo para que a tela não fique confusa.

A apresentação do jogo para o usuário é um fator importante a ser considerado. É a chamda "UX" (*User eXperience*) e será levada em consideração nos critérios de correção do TP.

Ao final do jogo, seu programa deverá anunciar o vencedor, o escore de partidas atualizado e se o jogador quer jogar ou não outra partida. No caso de o jogador não querer jogar mais, seu programa dará o escore final de partidas jogadas e uma mensagem de fim de jogo.

Documentação dos Trabalhos Práticos

Cada trabalho prático será corrigido levando-se em conta as *Recomendções para o Desenvolvimento de Programas*, apresentadas abaixo e os seguintes itens:[1]

Documentação

A documentação do programa do Trabalho Prático deve ser apresentada em três páginas, no máximo. Ela é como um pequeno *artigo* que explica o que o programa faz, como faz, e apresenta conclusões obtidas sobre o Trabalho Prático. Além disso, você incluirá a lsitagem do programa fonte, à parte, em um arquivo `.py`. A documentação é um documento à parte e não deve ser escrita no programa fonte.
A documentação a ser entregue deve conter pelo menos:

- *Descrição sucinta sobre o desenvolvimento do trabalho.*

  Uma explicação sobre as decisões de implementação tomadas, uma visão geral do funcionamento do programa e algoritmo em alto nível.

- *Descrição das funções e sua inter-dependência.*

  Uma breve descrição de cada função bem como um diagrama, por exemplo, mostrando a relação de dependência entre elas.

- *Descrição do formato de entrada dos dados.*

  Uma descrição simples e clara dizendo quais são os dados de entrada e como o programa irá recebê-los. Por exemplo:

  > "A entrada para o programa consiste de um conjunto de descrição dos edifícios. Em cada linha haverá somente uma descrição. Cada descrição é composta por três números inteiros separados por um ou mais brancos na seguinte ordem: coordenada esquerda do edifício, altura do edifício, coordenada direita do edifício."

- *Descrição do formato de saída dos dados.*

  Uma descrição simples e clara dizendo como o programa apresentará os resultados ao usuário. Por exemplo:

  > "O programa irá gerar uma seqüência de números representando a linha do horizonte. Números que estiverem nas posições ímpares representam coordenadas e números nas posições pares alturas."

- *Explicações sobre como utilizar o programa.*

  Por exemplo:

  > "Para executar o programa da linha do horizonte, digite na linha de comando: `lh.exe arqIn arqOut arqErro`, onde `arqIn` é o arquivo que contém os dados de entrada, ..."

[1] Baseado em notas de um ex-monitor de Ciência da Computação.

- *Testes.*

  Procure fazer testes significativos e relevantes como, por exemplo, casos de exceções. A listagem dos testes deve conter os dados recebidos pelo programa (dados de entrada) e os resultados apresentados (dados de saída). Comente os testes feitos.

- *Conclusão*

  Conclua o trabalho dizendo o que você achou do Trabalho Prático para seu aprendizado da arte de programação, citando dificuldades e prazeres na execução do mesmo.

- *Listagem do programa fonte.*

  Ver na sub-seção abaixo como apresentar a listagem do programa fonte.

Programa fonte

Procure observar os seguintes aspectos no seu programa fonte, a ser apresentado em arquivo prprio `.py`:

- *Modularidade.* Não faça programas de uma calsse só. Divida as tarefas a serem executadas em calsses diferentes.
- *Comentários.* Veja o item "Comentários" na seção de recomendações abaixo.
- *Indentação.* Veja o item "Indentação" na seção de recomendações abaixo.
- *Nomes de variáveis.* Veja o item "Nomes de variáveis" na seção de recomendações abaixo.

Apresentação

A apresentação do trabalho é importante. **Hoje**, o usuário ou cliente do seu TRABALHO é o professor ou o monitor do curso. **Amanhã** será o seu chefe ou um cliente para o qual você estará desenvolvendo um produto.
Coloque toda a documentação num só documento, utilizando papel do mesmo tipo, grampeado e sem capas plásticas. Verifique se a impressora está funcionando bem. Nunca rasure a listagem do programa ou os resultados de testes.

Recomendações para o Desenvolvimento de Programas[2]

Programas devem ser feitos para serem lidos por seres humanos.

Tenha em mente que seus programas deverão ser lidos e entendidos por outras pessoas (e/ou por você mesmo), de tal forma que possam ser corrigidos, modificados ou receber manutenção.

[2]Baseado em notas de aula preparadas pelos Professores Mariza Bigonha e Newton Vieira que, por sua vez, se baseou no livro "Program Style, Design, Efficiency, Debugging, and Testing" de D. Van Tassel, segunda edição, Prentice-Hall, 1978.

Escreva os comentários no momento que estiver escrevendo o algoritmo.

Um programa mal documentado é um dos piores erros que um programador pode cometer, e o sinal de um amador (mesmo com 10 anos de experiência).
O melhor momento para se escrever os comentários é aquele em que o programador tem maior intimidade com o algoritmo, ou seja, durante a sua confecção.

Os comentários devem acrescentar alguma coisa de útil, não apenas frasear o código.

O código nos diz *como* um certo problema está sendo resolvido. Os comentários deverão dizer *o que* está sendo feito em cada passo.

Use comentários no prólogo.

É muito importante a colocação de comentários no prólogo de um programa ou de cada módulo, para explicar o que ele faz e fornecer instruções para seu uso. Poderiam ser colocados comentários dos seguintes tipos:

- o que faz o programa ou módulo;
- como chamá-lo ou utilizá-lo;
- significado dos parâmetros, variáveis de entrada, de saída e variáveis mais importantes;
- arquivos utilizados;
- outros módulos utilizados;
- métodos especiais utilizados, com referências nas quais possa se encontrar mais informações;
- avaliação do tempo de processamento e memória requeridos;
- autor e data de escrita e última atualização;
- etc.

Utilize espaços em branco para melhorar a legibilidade.

Espaços em branco são valiosos para melhorar a aparência de um programa. Por exemplo:

- deixar uma linha em branco entre as declarações e o corpo do programa;
- deixar uma linha em branco antes e depois dos comentários;
- separar grupos de comandos que executam funções lógicas distintas por uma ou mais linhas em branco;

- utilizar brancos para indicar precedência de operadores; ao invés de `A+B * C` é bem mais legível a forma `A + B*C`;
- etc.

Escolha nomes representativos.

Os nomes de constantes, tipos, variáveis, procedimentos, funções, etc. devem identificar o melhor possível o que representam. Por exemplo, `X := Y + Z` é muito menos claro que `Preco := Custo + Lucro`.

Um comando por linha é suficiente.

A utilização de vários comandos por linha é prejudicial por vários motivos, dentre eles destacam-se o fato do programa tornar-se mais ilegível e ficar mais difícil de ser depurado. Exemplo (Pascal):

```
A:=14.2;for i:=1 to 10 do begin X[i]:=0;k:=i*k;Y[i]:=k;end;
```

O mesmo trecho poderia ser escrito assim:

```
A := 14.2;
for i:=1 to 10 do
begin
  X[i] := 0;
  k := i*k;
  Y[i] := k;
end;
```

Utilize parênteses para aumentar a legibilidade e prevenir-se contra erros.

Exemplos:

Com poucos parênteses	Com parênteses extras
`A*B*C/(D*E*F)`	`(A*B*C)/(D*E*F)`
`A*B/C*D/E*F`	`(A*B*D*F)/(C*E)`
`A/B/C`	`(A/B)/C`
`X >= Y or Q`	`(X >= Y) or Q`
`A+B < C`	`(A + B) < C`

Utilize "indentação" para mostrar a estrutura lógica do programa.

A indentação não deve ser feita de forma caótica, mas seguindo padrões razoáveis. Por exemplo, ao invés de:

```
if x < y then if v < w
then j := x
```

```
else if x < v then if y < w then j := y
     else j := v
else begin if x < w
then j := w
end else j := x + w;
```

uma forma mais razoável é (existe alguma forma de tornar o seguinte trecho mais simples?):

```
if x < y
then if v < w
     then j := x
     else if x < v
          then if y < w
               then j := y
               else j := v
          else begin
                 if x < w
                 then j := w
               end
else j := x + w;
```

Crie algumas regras básicas de indentação e procure seguí-las ao escrever um programa.

Lembre-se: a única parte da documentação que vai estar sempre em dia é o código do programa.

O programa deve ser mantido sempre legível, mesmo depois de quaisquer correções e/ou modificações.

Não comece a desenvolver o programa antes que o problema esteja bem definido.

Uma das partes mais importantes da documentação é a especificação do problema usando alguma notação: escrita, gráfica, etc. O ato de *escrever* a especificação do problema é muito importante: ajuda na sua total compreensão e evita o esquecimento de detalhes relevantes.
É preferível realismo no início do que ter que efetuar mudanças quando o programa atinge a "puberdade".

Após começar o desenvolvimento do programa procure não mudar a especificação do problema.

A única razão para mudar a especificação de um problema após começar o desenvolvimento do programa deve ser para corrigir algum erro de especificação. Se a especificação for mudada constantemente o programa nunca ficará pronto, além de acarretar mais custos.

Obviamente, isto não significa que não se deve alterar um programa depois que ele estiver pronto. É muito importante ter em mente que a maior parte dos programas desenvolvidos sofrerá algum tipo de modificação no futuro. Por essa razão, programas devem ser desenvolvidos de tal forma a minimizar o impacto introduzido pelas modificações.

A codificação deve ser feita de forma tão simples quanto possível.

Código complexo, sofisticado ou não usual atrapalha a legibilidade, depuração e modificação.

Estabeleça objetivos realistas o mais cedo possível.

Todo projeto de programação usualmente tem alguns objetivos que devem ser estabelecidos e colocados no papel. Tais objetivos variam naturalmente de projeto para projeto. Alguns itens a serem considerados são:

- data de término;
- facilidade de uso;
- nível de confiabilidade (difícil de ser estimado);
- limites de memória e tempo de execução;
- generalidade;
- etc.

Quando se trabalha em equipe, o estabelecimento explícito dos objetivos irá garantir que todos trabalhem para se alcançar os mesmos objetivos.

Desenvolva cuidadosamente o programa, a partir do algoritmo, usando a técnica de projeto como refinamento sucessivo.

Escrever diretamente qualquer código pode produzir uma barreira psicológica que poderá inibir futuros melhoramentos. Use alguma técnica de desenvolvimento de programas como a técnica de refinamento sucessivo.
Erros encontrados durante o desenvolvimento são relativamente fáceis de serem corrigidos comparados com erros encontrados durante os testes.

Um programa modesto que funciona é mais útil que um super-ambicioso que nunca funciona.

Muitas vezes é tentador fazer um programa com várias opções e melhorias. Mas nada adianta se esse programa nunca fica pronto para ser executado. Lembre-se que você não vai ter todo o tempo do mundo para escrever o seu programa. Por essa razão seja realista ao decidir o que fazer. Uma boa estratégia é ter uma versão simples que funciona e faz o que foi pedido e se houver tempo introduzir novas opções e melhorias.

Selecione os algoritmos cuidadosamente.

Existem algumas regras que você deve procurar seguir ao fazer um programa:

- Não reprograme um programa ou procedimento ou função que está pronto. A solução, no entanto, não é utilizar o primeiro algoritmo disponível. É muito importante que se entenda o módulo escolhido, ou seja, como funciona o algoritmo utilizado, quantidade de tempo e espaço necessários à execução desse módulo, etc.

  O que está sendo dito é que ninguém *inventa* todos os algoritmos que precisa utilizar em Ciência da Computação. O importante é saber e entender muito bem os algoritmos necessários à solução do seu problema.

- Não codifique o primeiro algoritmo que lhe venha à mente. O melhor algoritmo a ser utilizado depende dentre outros fatores da aplicação onde será utilizado. Por essa razão procure conhecer os possíveis algoritmos que podem ser usados na solução do problema. Só assim você terá certeza que escolheu um bom algoritmo ou não.

  A bibliografia citada apresenta vários livros na área de projeto de algoritmos.

Escolha a representação dos dados adequada ao problema.

Um boa representação dos dados pode eliminar páginas de código. A representação dos dados deve ser escolhida com base nas *principais operações* que serão feitas. Veja a bibliografia citada para o estudo de diferentes algoritmos e estruturas de dados.

Utilize uma linguagem de programação adequada.

Se a linguagem de programação não é adequada ao problema a ser resolvido, certamente surgirão problemas na programação, na eficiência e na depuração do programa.

Evite que o programa seja dependente de dados particulares.

Um programa dependente de dados particulares, além de ser mais restrito, dificulta mudanças. Um exemplo típico de dependência de dados é na manipulação de vetores. Por exemplo,

```
for i:=1 to 25 do
  readln(entrada, A[i]);
Soma := 0;
for i:=1 to 25 do
  Soma := Soma + A[i];
```

O trecho de programa acima só funciona para um vetor com 25 elementos. No caso de se desejar manipular um número diferente de elementos, deve-se percorrer o programa modificando cada 25, o que é inconveniente. Além do mais, pode-se saltar uma das constantes a ser modificada. E mais, pode-se modificar uma constante com o mesmo valor que não se deveria modificar. Uma solução melhor seria:

```
const N = 25;
. . .
for i:=1 to N do
  readln(entrada, A[i]);
Soma := 0;
for i:=1 to N do
  Soma := Soma + A[i];
```

Procure utilizar constantes, definir tipos, utilizar tipos abstratos de dados para evitar um problema similar ao apresentado acima.

Os formatos de entrada e saída devem ser bem legíveis.

Quando se projeta os formatos de entrada e saída deve-se ter sempre em mente o usuário. É importante definir uma ordem de variáveis e formatos de dados para entrada que sejam os mais naturais para o usuário. Isto evitará erros e facilitará o uso do programa. Por exemplo, o formato `:10` para leitura de quatro variáveis inteiras é bem mais razoável que o formato `:7, :8, :9, :8` para as mesmas quatro variáveis inteiras.
A saída deve ser bem legível, sem que haja necessidade de se consultar o código do programa.
A facilidade de uso (entrada) e relatórios atraentes (saída) serão os itens que, em última instância, determinarão o julgamento do usuário quanto à capacidade do programador.

Se o programa não funciona, não interessa sua eficiência.

Um programa super eficiente, mas não confiável, dificilmente pode ser convertido em um programa confiável. Mas um programa confiável e bem estruturado, mesmo que ineficiente, pode ser otimizado.
A legibilidade e', usualmente, mais importante que a eficiência. A legibilidade facilita o entendimento, permite possíveis modificações, inclusive a introdução de eficiência.

Antes de otimizar, procure saber quão eficiente o programa precisa ser.

A eficiência depende da aplicação, sendo mais relevante em alguns casos do quem em outros. Preferencialmente, o nível de eficiência que se quer de um programa deve constar na sua especificação.
Em geral, programas utilizados com muita freqüência devem ser mais eficientes do que os utilizados raramente.