08

Usando recursão para solucionar problema de distância.

Destruindo os fantasmas: o mundo da recursão

Destruindo os fantasmas

Nosso próximo passo é colocar uma bomba no meio jogo: quando o usuário passa por ela, ele detona os fantasmas próximos. Primeiro criamos um novo mapa, contendo as bombas (`*`) e mais fantasmas, nosso `mapa3.txt` :

```
XXXXXXXXXXXXXXXXX
X   FFF  X      X
X X XX X X X XX X
X X X* X     X  X
X   X XXXX XXX XX
  X     XX  X   X
 XXX XX XXX X X X
  X         X X X
X   X X XXX FFF X
X X     XXX XXX X
X X XXX   X XXX X
X  HXXX X X   X X
XXX*FXXFX   X   X
```

Passamos a carregar o terceiro mapa:

```
    mapa = le_mapa(3)
```

Primeiro, caso um fantasma entre em contato com uma bomba, ele simplesmente come ela, e nosso herói tem menos chances de ganhar o jogo. Portanto preste atenção, o jogador deve correr atrás das bombas o mais rápido possível.

Já quando o nosso herói encosta em uma bomba, devemos explodir quatro posições para a direita de onde a bomba está. Qualquer fantasma na região deve desaparecer, assim como muros:

```
def joga(nome)
# ...

        heroi.remove_do mapa
        if mapa[nova_posicao.linha][nova_posicao.coluna] == "*"
            mapa[nova_posicao.linha][nova_posicao.coluna + 1] = " "
            mapa[nova_posicao.linha][nova_posicao.coluna + 2] = " "
            mapa[nova_posicao.linha][nova_posicao.coluna + 3] = " "
            mapa[nova_posicao.linha][nova_posicao.coluna + 4] = " "
        end
        nova_posicao.coloca_no mapa
# ...
end
```

Uma implementação inicial, mas bem ruim. Obviamente podemos trocar por um `for`:

```
        heroi.remove_do mapa
        if mapa[nova_posicao.linha][nova_posicao.coluna] == "*"
            for direita in 1..4
                mapa[nova_posicao.linha][nova_posicao.coluna + direita] = " "
            end
        end
        nova_posicao.coloca_no mapa
```

Testamos nosso jogo com o terceiro mapa, andando para baixo, e vemos o resultado, quando os dois fantasmas e os muros são destruídos:

```
...
X X XXX   X XXX X
X  HXXX X X   X X
XXX*FXXFX   X   X
Para onde deseja ir?
...
X X XXX   X XXX X
X   XXX X X   X X
XXXH    X   X   X
```

Andando para a direita

Podemos extrair uma função para este trecho do código, algo que remova tudo do mapa a partir desta posição, por quatro casas. Invocamos ela:

```
        heroi.remove_do mapa
        if mapa[nova_posicao.linha][nova_posicao.coluna] == "*"
            remove mapa, nova_posicao, 4
        end
        nova_posicao.coloca_no mapa
```

E extraímos seu código:

```
def remove mapa, posicao, quantidade
    for direita in 1..quantidade
        mapa[posicao.linha][posicao.coluna + direita] = " "
    end
end
```

O resultado ainda é o mesmo. Mas repare que está meio estranho essa história de toda vez somar um número cada vez maior. Ao invés disso poderíamos falar para a posição andar para a direita. Lembra que nosso herói sabe se

movimentar? Ele mesmo tem o método `calcula_nova_posicao`:

```
def remove mapa, posicao, quantidade
    for direita in 1..quantidade
        posicao = posicao.calcula_nova_posicao "D"
        mapa[posicao.linha][posicao.coluna] = " "
    end
end
```

Mas, Guilherme, está um pouco estranho. Passar `D` como argumento? Estamos programando orientado a `String` ou a objetos? Seria mais educado fazer:

```
def remove mapa, posicao, quantidade
    for direita in 1..quantidade
        posicao = posicao.direita
        mapa[posicao.linha][posicao.coluna] = " "
    end
end
```

E criamos o método `direita`:

```
class Heroi
    # ...

    def direita
        calcula_nova_posicao "D"
    end
end
```

Recursão infinita

Mas repare que a lógica de nossa função ainda é razoavelmente complexa. Note que o que o laço faz é passar com a `direita` do `1` até `4`, isto é, `posicao.coluna+1` até `posicao.coluna+4`. Estamos fazendo um laço, mudando a posição e limpando um trecho do mapa. Não seria possível simplificar nossa função, removendo esse laço?

```
def remove mapa, posicao, quantidade
    for direita in 1..quantidade
        posicao = posicao.direita
        mapa[posicao.linha][posicao.coluna] = " "
    end
end
```

O que seria de nossa função se ela somente removesse a primeira posição da direita?

```
def remove mapa, posicao, quantidade
    posicao = posicao.direita
    mapa[posicao.linha][posicao.coluna] = " "
end
```

Não seria o suficiente, pois ainda temos que remover mais posições a direita, claro. Quantas posições faltam remover? Três. Isto é, `quantidade - 1`. Invoquemos então a própria função novamente, agora passando `quantidade - 1`, isto é, `3`:

```
def remove mapa, posicao, quantidade
    posicao = posicao.direita
    mapa[posicao.linha][posicao.coluna] = " "
    remove mapa, posicao, quantidade - 1
end
```

O que acontece agora? A função será invocada para a segunda posição a direita, com quantidade valendo 2, a próxima posição a direita com quantidade valendo 1, e a última posição a direita com quantidade valendo 0, mas ela continua sendo invocada eternamente!

Vejamos como funciona, vamos simular nossa pilha de execução. Minhas linhas da função remove são como a seguir:

```
def remove mapa, posicao, quantidade
    posicao = posicao.direita                    # linha 109
    mapa[posicao.linha][posicao.coluna] = " "    # linha 110
    remove mapa, posicao, quantidade - 1         # linha 111
end
```

Primeiro chamamos a função `remove` com o `mapa`, posição por exemplo `5, 7` e `quantidade` valendo `4`:

```
fogefoge.rb:remove:109 (mapa =..., posicao = 5,7, quantidade = 4)
```

Ao terminar a execução da linha 109, indo para a 110, temos que a posição foi alterada:

```
fogefoge.rb:remove:110 (mapa =..., posicao = 5,8, quantidade = 4)
```

Portanto limpamos a posição `5,8` do mapa, o primeiro quadrado a direita. Agora executamos a linha 111, que invoca a função `remove` com `quantidade` valendo 3:

```
fogefoge.rb:remove:109 (mapa =..., posicao = 5,8, quantidade = 3)
fogefoge.rb:remove:111 (mapa =..., posicao = 5,8, quantidade = 4)
```

Primeiro ele move a posição para a direita:

```
fogefoge.rb:remove:110 (mapa =..., posicao = 5,9, quantidade = 3)
fogefoge.rb:remove:111 (mapa =..., posicao = 5,8, quantidade = 4)
```

Limpa a posição `5,9` e depois chama a função `remove` novamente, empilhando mais uma vez a função na pilha:

```
fogefoge.rb:remove:109 (mapa =..., posicao = 5,9, quantidade = 2)
fogefoge.rb:remove:111 (mapa =..., posicao = 5,9, quantidade = 3)
fogefoge.rb:remove:111 (mapa =..., posicao = 5,8, quantidade = 4)
```

Mais uma vez será movido para a direita:

```
fogefoge.rb:remove:110 (mapa =..., posicao = 5,10, quantidade = 2)
fogefoge.rb:remove:111 (mapa =..., posicao = 5,9, quantidade = 3)
fogefoge.rb:remove:111 (mapa =..., posicao = 5,8, quantidade = 4)
```

O mapa na posição `5,10` é limpado, e invocamos a função novamente:

```
fogefoge.rb:remove:109 (mapa =..., posicao = 5,10, quantidade = 1)
fogefoge.rb:remove:111 (mapa =..., posicao = 5,10, quantidade = 2)
fogefoge.rb:remove:111 (mapa =..., posicao = 5,9, quantidade = 3)
fogefoge.rb:remove:111 (mapa =..., posicao = 5,8, quantidade = 4)
```

O mesmo processo ocorre para `5,11` :

```
fogefoge.rb:remove:109 (mapa =..., posicao = 5,11, quantidade = 0)
fogefoge.rb:remove:111 (mapa =..., posicao = 5,11, quantidade = 1)
fogefoge.rb:remove:111 (mapa =..., posicao = 5,10, quantidade = 2)
fogefoge.rb:remove:111 (mapa =..., posicao = 5,9, quantidade = 3)
fogefoge.rb:remove:111 (mapa =..., posicao = 5,8, quantidade = 4)
```

E o fluxo continua...

```
fogefoge.rb:remove:109 (mapa =..., posicao = ..., quantidade = -576)
...
fogefoge.rb:remove:111 (mapa =..., posicao = 5,13, quantidade = -1)
fogefoge.rb:remove:111 (mapa =..., posicao = 5,12, quantidade = 0)
fogefoge.rb:remove:111 (mapa =..., posicao = 5,11, quantidade = 1)
fogefoge.rb:remove:111 (mapa =..., posicao = 5,10, quantidade = 2)
fogefoge.rb:remove:111 (mapa =..., posicao = 5,9, quantidade = 3)
fogefoge.rb:remove:111 (mapa =..., posicao = 5,8, quantidade = 4)
```

Podemos ver isso acontecer rodando o jogo:

```
heroi.rb:4: stack level too deep (SystemStackError)
```

Opa. O que aconteceu? O nível de profundidade foi muito grande, pois chamamos uma função, que chamou ela mesma, que chamou ela mesma, que chamou ela mesma, eternamente. Mas claro, existe algum limite - como tudo em um computador - para o tamanho da pilha, e esse limite alguma hora foi alcançado. O programa parou.

Estamos fazendo com que nossa função chame ela mesma, uma técnica chamada de ::recursão::.

A base da recursão

O problema de uma recursão infinita é que uma hora a pilha de execução estoura, e o programa para. Queremos então definir um ponto de parada dela, no nosso caso só queremos executar nossa função enquanto a quantidade for maior que zero, isto é:

```
def remove mapa, posicao, quantidade
    if quantidade == 0
        return
    end
    posicao = posicao.direita
    mapa[posicao.linha][posicao.coluna] = " "
    remove mapa, posicao, quantidade - 1
end
```

Unless: trava de segurança (safeguard)

Outra maneira de escrever esse código seria usando o unless de uma linha

```
def remove mapa, posicao, quantidade
    return unless quantidade > 0
    posicao = posicao.direita
    mapa[posicao.linha][posicao.coluna] = " "
    remove mapa, posicao, quantidade - 1
end
```

Temos uma inversão, uma negação, o que pode ficar confuso: não execute isso se for maior do que zero.

Esse ponto de parada da recursão é o que chamamos de base da recursão. Quando ela chega naquele ponto, ela para. Rodando agora nossa aplicação teríamos o seguinte ::stack trace:::

```
fogefoge.rb:remove:109 (mapa =..., posicao = 5,12, quantidade = 0)
fogefoge.rb:remove:112 (mapa =..., posicao = 5,11, quantidade = 1)
fogefoge.rb:remove:112 (mapa =..., posicao = 5,10, quantidade = 2)
```

```
fogefoge.rb:remove:112 (mapa =..., posicao = 5,9, quantidade = 3)
fogefoge.rb:remove:112 (mapa =..., posicao = 5,8, quantidade = 4)
```

E nesse instante, na nova linha `109` ele encontra que quantidade vale `0` e para. Retornando de cada uma das funções e continuando o jogo normalmente. Podemos testá-lo e ver que realmente funciona:

```
...
X X XXX   X XXX X
X  HXXX X X   X X
XXX*FXXFX   X   X
...
X X XXX   X XXX X
X   XXX X X   X X
XXXH    X   X   X
```

Recursão é o ato de uma função chamar ela mesma. Base da recursão é o critério onde essa recursão para, caso contrário ela executaria indeterminadamente - o que poderia causar um estouro da pilha de execução.

Repare como nosso código final não possui mais o laço, utiliza a recursão e a pilha de execução para que o acumulador (`quantidade`) conte quantas casas foram limpadas no mapa:

```
def remove mapa, posicao, quantidade
    if quantidade == 0
        return
    end
    posicao = posicao.direita
    mapa[posicao.linha][posicao.coluna] = " "
    remove mapa, posicao, quantidade - 1
end
```

Repare que fomos capazes de executar a recursão e a iteração (o laço) para atingir o mesmo resultado.

Base da recursão: distância quatro ou muro

Na prática nossa bomba é forte demais, queremos que ela seja incapaz de explodir um muro. É muito incomum que um jogo permita esse tipo de explosão, portanto não podemos continuar se a posição atual for um muro.

Na implementação antiga, de um laço, poderíamos utilizar uma condição e um `break` :

```
def remove mapa, posicao, quantidade
    for direita in 1..quantidade
        posicao = posicao.direita
        if mapa[posicao.linha][posicao.coluna] == "X"
            break
        end
        mapa[posicao.linha][posicao.coluna] = " "
    end
end
```

Na nossa versão atual, recursiva, da função `remove`, basta adicionarmos uma nova base, se tem muro, para:

```
def remove mapa, posicao, quantidade
    if quantidade == 0
        return
    end
    posicao = posicao.direita
    if mapa[posicao.linha][posicao.coluna] == "X"
        return
    end
    mapa[posicao.linha][posicao.coluna] = " "
    remove mapa, posicao, quantidade - 1
end
```

Resultando em:

```
X X XXX   X XXX X
X  HXXX X X   X X
XXX*FXXFX   X   X
...
X X XXX   X XXX X
X   XXXFX X   X X
XXXH XX X   X   X
```

Podemos extrair agora uma função `executa_remocao`:

```
def executa_remocao mapa, posicao, quantidade
    if mapa[posicao.linha][posicao.coluna] == "X"
        return
    end
    mapa[posicao.linha][posicao.coluna] = " "
    remove mapa, posicao, quantidade - 1
end

def remove mapa, posicao, quantidade
    if quantidade == 0
        return
    end
    executa_remocao mapa, posicao.direita, quantidade
end
```

Recursão para todos os lados: busca em profundidade

Por mais que nossa bomba respeite muros, nosso efeito ainda é bem estranho. Tradicionalmente em jogos uma bomba explode para todos os lados ao mesmo tempo, andando, por exemplo, 4 casas no máximo para qualquer direção, que é o alcance dela.

Dado o mapa abaixo, o `mapa4.txt` :

```
XXXXXX
XHXXXX
X*FX X
XXFX X
XFFFFX
XXXXXX
```

A explosão ::anda:: quatro casas, de todas as maneiras possíveis. Note então o quão longe ela deveria conseguir chegar, somente até os fantasmas marcado 4:

```
XXXXXX
XHXXXX
X*1X X
XX2X X
X4345X
XXXXXX
```

Em outras palavras somente um fantasma sobreviveria. Como calcular isso? Queremos andar quatro de distância, por ::todos os caminhos possíveis::, a partir de nossa bomba.

Uma solução inicial, que temos que tomar um cuidado, não podemos fazer dois `for` , de `-4` a `+4` :

```
for movimento_linha in -4..4
    for movimento_coluna in -4..4
        linha = nova_posicao.linha + movimento_linha
        coluna = nova_posicao.coluna + movimento_coluna
        mapa[linha][coluna] = " "
    end
end
```

O movimento de nosso herói resultaria em:

```
XXXXXX
X XXXX
XH X X
XX X X
X    X
XXXXXX
```

Implementação completamente errada. Explodimos para todos os lados. Não é essa a descrição da explosão. Devemos percorrer todos os corredores de quatro de distância do ponto de origem, e limpar todos eles, nada mais.

Vamos olhar para nossa implementação recursiva:

```
def remove mapa, posicao, quantidade
    if quantidade == 0
        return
    end
    executa_remocao mapa, posicao.direita, quantidade
end
```

Ao invés de tentarmos explodir somente a direita, tentamos as quatro direções, isto é, tentamos andar quatro de distância em todos os caminhos possíveis: estamos buscando todas as casas que conseguimos alcançar com quatro passos.

```
def remove mapa, posicao, quantidade
    if quantidade == 0
        return
    end
    executa_remocao mapa, posicao.direita, quantidade
    executa_remocao mapa, posicao.esquerda, quantidade
    executa_remocao mapa, posicao.cima, quantidade
    executa_remocao mapa, posicao.baixo, quantidade
end
```

Claro, adicionamos as funções de direção ao herói:

```
class Heroi
    # ...
    def cima
        calcula_nova_posicao "W"
    end
    def esquerda
        calcula_nova_posicao "A"
    end
    def baixo
        calcula_nova_posicao "S"
    end
end
```

Carregamos o quarto mapa:

```
[code ruby]
def joga(nome)
    mapa = le_mapa(4)
    # ...
end
```

E agora tentamos rodar:

```
XXXXXX
XHXXXX
```

```
X*FX X
XXFX X
XFFFFX
XXXXXX
Para onde deseja ir?
S
XXXXXX
X XXXX
XH X X
XX XFX
X    X
XXXXXX
```

Resumindo

Vimos nesse capítulo como implementar o algoritmo de destruição de um fantasma, para isso refatoramos mais nosso código, implementamos um laço através de um loop e de invocar a própria função, a recursão.

Encontramos a base da recursão quando percebemos que a pilha de execução estourava a medida que entrava em um laço infinito (ela se invocava até estourar). E com a recursão foi muito simples permitir andar quatro passos para todos os lados: foi possível buscar todos os quadrados do mapa que estão a até quatro passos distância, uma busca chamada ::busca em profundidade::.