# Teste de Funções por Partição do Espaço de Entrada

Universidade de Lisboa
Faculdade de Ciências
Departamento de Informática
Licenciatura em Tecnologias da Informação

Vasco Thudichum Vasconcelos

Fevereiro 2017*

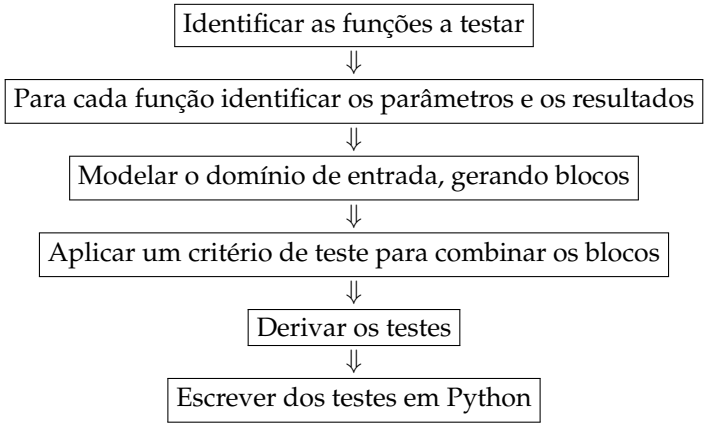
## Introdução

A atividade de teste fundamenta-se em última instância na escolha de parâmetros para correr o software em teste. O método de teste por partição do espaço de entrada parte da observação que o espaço de entrada (ou o conjunto de partida) do artefato em teste é definido em termos dos possíveis valores que os parâmetros podem tomar. Este espaço é *particionado* em *blocos* que assumimos conterem valores igualmente úteis do ponto de vista da atividade de teste.

De um modo geral há muitos artefatos de software que podem ser testados. No caso de Programação II vamos testar apenas funções. O espaço de entrada de uma função contém não só os parâmetros que aparecem explícitos na definição da função, mas também valores de variáveis globais, para além de todo o ambiente de execução do programa. De todos estes componentes do espaço de entrada vamos considerar unicamente os parâmetros de entrada das funções, tratando assim apenas funções que não acedem a variáveis globais.

Assim sendo, para cada função e considerando exclusivamente os seus parâmetros, identificamos uma série de partições do seu espaço de entrada. A ideia subjacente à partição do espaço de entrada é a de que todos os valores num dado bloco são igualmente bons para efeitos de teste. Seleccionamos então valores pertencentes a cada um dos blocos da partição e, finalmente, usamos estes valores para gerar testes. O processo de geração de testes por partição do espaço de entrada pode ser resumido deste modo:

*Pequenos ajustes em fevereiro 2021.




Este método de teste traz várias vantagens:

- Requer pouco treino;
- O engenheiro de testes não precisa de perceber a implementação da função;
- Os testes podem ser reutilizados em caso de alteração na implementação.

## Partição de um conjunto

Comecemos por relembrar o conceito de *partição*. Uma partição de um conjunto $X$ é uma coleção de subconjuntos de $X$ tais que cada elemento de $X$ pertence a um e um só destes subconjuntos. Dito de outro modo, uma família de conjuntos $P$ é uma partição de $X$ se as seguintes condições se verificarem simultaneamente:

1. A família $P$ não contém o conjunto vazio,
2. A união dos conjuntos em $P$ forma o conjunto $X$ (isto é, os conjuntos em $P$ cobrem $X$), e
3. A interseção de dois conjuntos distintos em $P$ é vazia (isto é, os conjuntos em $P$ são disjuntos dois a dois).

A cada um dos conjuntos em $P$ chamamos um *bloco*. Alguns exemplos:

- Consideremos o espaço dos números inteiros. A família de conjuntos constituída por "números negativos" e "números positivos" *não* é uma partição porque a sua reunião não cobre o espaço: o número zero não pertence a nenhum dos conjuntos. Já a família de conjuntos constituída pelos conjuntos "números negativos" e "números não negativos" é uma partição, como também o é a família { "números negativos", "0", "números positivos"}.

Figura 1: Um conjunto de selos particionado em blocos: nenhum selo pertence a dois blocos, nenhum bloco está vazio e todos os selos estão em algum bloco [2].

- Consideremos o espaço das listas de números inteiros. A família de conjuntos {"listas ordenadas por ordem crescente", "listas ordenadas por ordem decrescente", "listas cujos elementos não estão ordenados"} *não* é uma partição porque os conjuntos não são disjuntos dois a dois: as listas vazias pertencem aos três conjuntos e as listas com um elemento pertencem aos dois primeiros. Já a família de conjuntos {"listas ordenadas por ordem crescente", "listas não ordenadas por ordem crescente"} é uma partição.[1]
- Consideremos agora o espaço de todas as listas. A família de conjuntos {"listas de comprimento negativo", "listas de comprimento não negativo"} *não* é uma partição, apesar dos dois conjuntos cobrirem o espaço e de serem disjuntos: o conjunto "listas de comprimento negativo" é vazio. Já a família de conjuntos {"listas vazias", "listas não vazias"} é uma partição.

## Modelação do espaço de entrada de uma função

A modelação do espaço de entrada de uma função é feita em três passos:

Identificar as características de uma função
⇓
Efectuar a partição de cada característica em blocos
⇓
Identificar um valor para cada bloco

As próximas três secções descrevem estes três passos.

[1] De notar que listas vazias e listas com um elemento estão ordenadas por definição.

## Identificação das características

Para gerar testes precisamos de partições. Dada uma função, como obter as partições necessárias? As partições são identificadas com base no *modelo do espaço de entrada*. Esta é a fase mais criativa do processo de teste. Um modelo do espaço de entrada descreve o espaço de entrada da função a um certo nível de abstração. O engenheiro de testes descreve a estrutura do espaço de entrada em termos de *características*. Cada característica dá origem a uma partição formada por um conjunto de bloco. Dos blocos retiramos depois valores para os testes.

Cada característica deve representar uma faceta do espaço de entrada da função que interessa explorar. Eis alguns exemplos:

- A característica (sinal) de um número: negativo, zero ou positivo;
- A cor do semáforo;
- O facto da lista estar ordenada;
- O número de chaves num dicionário.

Diferentes engenheiros de teste identificarão diferentes características e diferentes blocos para uma mesma função. Tal cria necessariamente alguma variância na qualidade dos testes resultantes. O método descrito neste documento tenta minorar esta variância, aumentado deste modo a qualidade do modelo do espaço de entrada.

O processo de construção do modelo do espaço de entrada de uma função começa com a identificação de todos os parâmetros que possam afetar o resultado de uma função. No caso de Programação II, estamos interessados apenas nos parâmetros das funções, e estes estão todos explícitos na assinatura da função. Por exemplo, para a função `membro` definida abaixo, os parâmetros são `lista` e `elemento`.

```
def membro (lista, elemento):
    """Um dado elemento ocorre numa dada lista?

    Args:
        lista (list): A lista onde procurar
        elemento (any): O elemento a procurar

    Returns:
        bool: True se o elemento está na lista; False
            caso contrário
    """
```

Com o propósito de identificar características podemos pensar na *interface* da função e na *funcionalidade* (isto é o *contrato*) desta. Vamos discutir as duas abordagens em seguida.

**Modelação do espaço de entrada baseado na interface** Neste caso olhamos apenas para a interface da função. A interface contém a assinatura da função. A assinatura de uma função Python é a primeira linha da função:

```
def membro (lista, elemento):
```

juntamente com alguma informação sobre o tipo dos parâmetros esperado. Tal pode ser inferido através do nome do parâmetro (`lista`, neste caso), ou olhando para o `doctest` que acompanha a função, tal como no exemplo acima.

No caso da função `membro`, procuramos características para a `lista` e para o `elemento`. O `doctest` diz-nos que o primeiro parâmetro é do tipo lista. Olhando *apenas para a interface* da função, podemos utilizar a característica mais comum para as listas:

- A lista está vazia.

Consideremos agora para o segundo parâmetro, `elemento`. Que características poderíamos considerar para este parâmetro? Olhando para o `doctest` da função concluímos que não há restrição alguma no que diz respeito ao elemento a procurar (podemos procurar elementos de qualquer tipo), pelo que não escolhemos nenhuma característica para o parâmetro.

Esta abordagem tem uma grande vantagem: é extremamente fácil identificar características. O fato de cada característica se limitar apenas a um parâmetro facilita muito o processo de criação de testes. A grande desvantagem é que nem toda a informação disponível é usada na modelação do espaço de entrada. Tal pode conduzir a um modelo deficiente, que conduzirá mais tarde a testes pouco efetivos.

Outro ponto fraco da análise isolada da interface da função é o fato de esta não tomar em consideração as possíveis relações entre os vários parâmetros e/ou o resultado esperado da função. A análise da interface tem de ser complementada com o estudo da funcionalidade da função.

**Modelação do espaço de entrada baseado na funcionalidade** Para modelar o espaço de entrada de uma função baseado na sua funcionalidade, munimo-nos do contrato da função. Este está geralmente descrito no `doctest`. Outras vezes está, infelizmente omisso, e tem de ser "inferido".

Pensando agora em termos da funcionalidade da função, podemos considerar características que relacionem os dois parâmetros. Por exemplo:

- O número de ocorrências de `elemento` na `lista`, e
- O `elemento` ocorre na primeira posição da `lista`.

A identificação de características é desenvolvida nos exemplos no final deste documento.

O método baseado na funcionalidade resulta em melhores casos de testes pois informação importante sobre o comportamento esperado da função é tomada em consideração. Outras das vantagens é que obriga à escrita do contrato antes da escrita dos testes. De fato, depois da escrita do contrato, duas atividades podem acontecer em paralelo:

- O engenheiro de testes desenvolve os testes para a função, e
- O engenheiro de software desenvolve o código da função.

**Modelação do comportamento excecional** Até agora concentrámo-nos na modelação do *domínio* das funções. Quais as características associadas ao comportamento de uma função *fora* do seu domínio? Na realidade nem sempre é possível modelar este comportamento por falta de informação. Consideremos a seguinte função:

```
"""A média dos elementos de uma lista não vazia

Args:
    l (list[number]): A lista

Returns:
    float: A média dos elementos da lista
"""
```

Neste caso o espaço de entrada da função é o conjunto de todas as listas e o domínio é o conjunto de todas as listas não vazias e de valores numéricos. É bom de ver que não temos informação sobre o comportamento da função para listas vazias e que, portanto, não conseguimos escrever testes neste caso.

Consideremos agora a seguinte variante da mesma função:

```
"""A média dos elementos de uma lista não vazia

Args:
    l (list[number]): A lista

Raises:
    ZeroDivisionError: Quando a lista é vazia

Returns:
    float: A média dos elementos da lista
"""
```

Apesar do espaço de entrada e do domínio se manterem, sabemos agora um pouco mais sobre o comportamento da função *fora* do seu domínio: sabemos que levanta uma exceção `ZeroDivisionError` no caso de listas vazias. Podemos então considerar a característica:

- A lista é vazia?

E que característica poderíamos associar a listas com elementos não numéricos? Estamos novamente em presença de valores fora do domínio da função. Como não temos informação sobre o comportamento da função neste caso, não geramos nenhuma característica, o que quer dizer que não vamos gerar testes com listas com valores não numéricos.

## Partição das características em blocos

Uma vez identificadas as características, chega a altura de particionar as características em famílias de conjuntos, denominados *blocos*. Não esquecer que:

- *Mais blocos* implicam mais testes, requerendo mais recursos (tempo para codificar e, mais tarde, para executar os testes), mas podem revelar mais falhas no programa, e que
- *Menos blocos* resultam em menos testes, e portanto numa poupança de recursos, podendo no entanto reduzir a eficácia dos testes.

Não esquecer também que os vários blocos devem constituir uma *partição do espaço de entrada*, isto é,

1. Os blocos não se devem intersetar,
2. Todos juntos devem cobrir o espaço na sua totalidade, e
3. Não deve haver nenhum bloco vazio.

Prosseguindo com o exemplo da função `membro`, tomemos a característica "número $n$ de ocorrências do elemento na lista". Os blocos $n > 1$ e $n > 2$ intersetam-se e como tal não podem ser tomados em conjunto. Os blocos $n = 0$ e $n > 1$ não cobrem o espaço total (que é dado pela inequação $n \geq 0$), e como tal temos de juntar a bloco $n = 1$.

De acordo com as características enunciadas acima, podemos identificar os seguintes blocos:

| Característica | Blocos |
|---|---|
| A lista está vazia | $v_1 =$ True, $v_2 =$ False |
| Número de ocorrências na lista | $n_1 = 0$, $n_2 = 1$, $n_3 > 1$ |
| Elemento ocorre na primeira posição da lista | $p_1 =$ True, $p_2 =$ False |

De notar que escolhemos três blocos para a característica "Número de ocorrências na lista", mas poderíamos ter escolhido um outro número (dois ou quatro, por exemplo).

## Identificação dos casos de teste

O último passo no nosso método consiste em identificar os casos de teste. A questão que se põe é "Como considerar as várias partições em simultâneo?" Por outras palavras, "Que combinações de blocos vamos usar para escolher valores?" Há muitos critérios para a escolha de combinações. Na disciplina de Programação II vamos apenas usar um: o da *cobertura de todas as combinações*.

No caso da função `membro` identificámos três características com blocos $\{v_1, v_2\}$, $\{n_1, n_2, n_3\}$ e $\{p_1, p_2\}$. Precisamos então de $12 =$

$2 \times 3 \times 2$ testes para cobrir todas as combinações possíveis. São eles: $(v_1, n_1, p_1)$, $(v_1, n_1, p_2)$, $(v_1, n_2, p_1)$, $(v_1, n_2, p_2)$, $(v_1, n_3, p_1)$, $(v_1, n_3, p_2)$, $(v_2, n_1, p_1)$, $(v_2, n_1, p_2)$, $(v_2, n_2, p_1)$, $(v_2, n_2, p_2)$, $(v_2, n_3, p_1)$ e $(v_2, n_3, p_2)$. Está fácil de ver que, de um modo geral, a utilização deste critério pode tornar-se impraticável quando tiverem sido identificadas mais do que 2 ou 3 características e respetivas partições.

Um ponto algo subtil sobre o método de partição do espaço de entrada é que algumas combinações de blocos são *inviáveis*. Por exemplo, no caso da função `membro`, não podemos combinar o bloco $v_1$ com o bloco $n_2$, porque precisaríamos de uma `lista` vazia que contivesse uma ocorrência do `elemento`. Do mesmo modo não podemos combinar a bloco $n_0$ com a bloco $p_2$, porque não conseguimos construir uma `lista` com zero ocorrências do `elemento`, onde este ocorra na primeira posição da lista.

Estamos agora em condições de identificar valores representativos para cada um dos blocos. Como foi dito anteriormente, *qualquer valor de um bloco é igualmente útil na geração do teste*, de modo que, uma vez identificados os blocos não precisamos de pensar muito na escolha dos valores de teste: qualquer valor que pertença ao bloco serve para efeitos de teste. Assim sendo, usando o critério de cobertura de todas as combinações construímos uma tabela com os blocos e os valores de teste.

| Características e blocos | | | Testes | |
|---|---|---|---|---|
| Vazia | Num. | Primeiro | Entrada | Resultado |
| True | 0 | True | Inviável | |
| True | 0 | False | `([],"c")` | `False` |
| True | 1 | True | Inviável | |
| True | 1 | False | Inviável | |
| True | >1 | True | Inviável | |
| True | >1 | False | Inviável | |
| False | 0 | True | Inviável | |
| False | 0 | False | `(["a","b"], "c")` | `False` |
| False | 1 | True | `(["c","a","b"], "c")` | `True` |
| False | 1 | False | `(["a","c","b"], "c")` | `True` |
| False | >1 | True | `(["c","a","c","b"], "c")` | `True` |
| False | >1 | False | `(["a","c","c","b"], "c")` | `True` |

Vemos então que, das doze combinações de testes identificadas para a função `membro`, seis são inviáveis, restando deste modo seis combinações.

## Teste de funções com `doctest`

O módulo `doctest`[2] procura pedaços de texto que se pareçam com sessões Python interativas e em seguida executa essas sessões para verificar se funcionam como mostrado. Para isso colocamos os testes dentro da *string* `doctest` do seguinte modo:

[2] doctest — Test interactive Python examples.

```
>>> membro([], "c")              # (v1,n1,p2)
False
```

Os testes identificados na tabela acima podem ser concretizados deste modo:

```
def membro (lista, elemento):
    """Um dado elemento ocorre numa dada lista?

    Args:
        lista (list): A lista onde procurar
        elemento (any): O elemento a procurar

    Returns:
        bool: True se o elemento está na lista; False
            caso contrário

    >>> membro([], "c")                # (v1,n1,p2)
    False
    >>> membro(["a","b"], "c")         # (v2,n1,p2)
    False
    >>> membro(["c","a","b"], "c")     # (v2,n2,p1)
    True
    >>> membro(["a","c","b"], "c")     # (v2,n2,p2)
    True
    >>> membro(["c","a","c","b"], "c") # (v2,n3,p1)
    True
    >>> membro(["a","c","c","b"], "c") # (v2,n3,p2)
    True
    """
```

No final da módulo colocamos as três linhas mágicas que permitem correr todos os testes de todas as funções quando o módulo é usado como programa principal:

```
if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

Se o módulo se chamar `M.py`, o comando

```
$ python M.py
```

corre os testes todos. O comando não exibe nada, a menos que um exemplo falhe, caso em que os exemplos com falhas e as causas das falhas são impressos no terminal. Para obter um relatório detalhado de todos os exemplos testados usar a opção `-v`:

```
$ python M.py -v
```

## Mais sobre identificação de características e de blocos

Eis algumas ideias que nos podem ajudar a escolher características e correspondentes blocos para funções Python.

- Para cada parâmetro numérico $x$, escolhemos a característica que identifica o sinal do número (negativo, zero, positivo). Desta característica extraímos três blocos: $x < 0$, $x = 0$ e $x > 0$.
- O item acima aplica-se para funções cujo contrato não restrinja o domínio do parâmetro. Se a pré-condição ditar que o parâmetro é, por exemplo, não negativo, então podemos pensar na característica "dimensão do número" e extrair três blocos: $x = 0$, $x = 1$ e $x > 1$.
- Se o contrato da função falar de um valor $n$ em especial, então podemos escolher três blocos: $x < n$, $x = n$ e $x > n$.
- Se a pré-condição atribuir limites $n$ e $m$ para o parâmetro $x$, então podemos escolher três blocos: $x = n$, $n < x < m$ e $x = m$.
- Para cada lista escolhemos a característica "lista vazia", da qual resultam dois blocos: lista vazia e lista não vazia.
- Para cada dicionário analisamos as suas várias entradas.
- Para cada tuplo analisamos as suas componentes.
- Para cada resultado possível da função procedemos como se de um parâmetro se tratasse.
- Tomamos também em consideração o nosso conhecimento da funcionalidade da função, bem como outras condições constantes no contrato da função.

## O método de partição do espaço de entrada em cinco passos

Em resumo, eis o método que vamos seguir, em cinco passos:

1. Identificar o espaço de entrada da função (que como vimos, se resume aos seus parâmetros);
2. *Modelar o espaço de entrada* através da identificação de uma ou mais *características* deste. Para além da assinatura da função, tomamos também em consideração o seu contrato (que determina o comportamento esperado);
3. Para cada característica, definir *blocos* que particionem o espaço;
4. Aplicar o critério de cobertura de todas as combinações, identificando deste modo as combinações de blocos para testes (e excluindo as combinações inviáveis);

5. Escolher valores para cada bloco de modo a obter os *inputs* para os testes (isto é, os casos de teste).

## Exemplos de casos comuns

Tendo visto o caso geral, vamos analisar alguns casos comuns.

### Parâmetro numérico único

```
def simetrico (x):
    """O simétrico de um número

    Args:
        x (number): O número

    Returns:
        number: O simétrico
    """
```

Sem mais restrições sobre o parâmetro `x` (para além do facto que este ter de ser um número) podemos considerar a característica mais comum dos números: o seu sinal (negativo, zero e positivo). Tal característica leva aos blocos $x < 0$, $x = 0$ e $x < 0$, que por sua vez ditam três testes simples.

```
def simetrico (x):
    """O simétrico de um número

    Args:
        x (number): O número

    Returns:
        number: O simétrico

    >>> simetrico(-2.7)
    2.7
    >>> simetrico(0)
    0
    >>> simetrico(7.8)
    -7.8
    """
```

### Parâmetro númerico com restrições

```
def factorial (n):
    """O factorial de um número

    Args:
```

```
        n (number): O número
    Pre:
        n >= 0
    Returns:
        number: O factorial de n
    """
```

Neste caso o espaço de entrada da função é o conjunto dos números inteiros e o domínio é o conjunto dos números não negativos. Podemos pensar na característica "dimensão do número", para a qual podemos gerar muitos blocos distintos. Por exemplo, os blocos $n = 0$, $n = 1$ e $n > 1$ ditam três testes.

```
def factorial (n):
    """O factorial de um número

    Args:
        n (number): O número
    Pre:
        n >= 0
    Returns:
        number: O factorial de n

    >>> factorial(0)
    1
    >>> factorial(1)
    1
    >>> factorial(5)
    120
    """
```

### Tomando em consideração a funcionalidade da função

```
def hexa(x):
    """A representação hexadecimal de um número

    Args:
        x (integer): O número a convert para hexadecimal
    Pre:
        0 <= x <= 15
    Returns:
        str: A representação hexa
    """
```

O espaço de entrada desta função é o conjunto dos números inteiros e o domínio é o conjunto dos números entre 0 e 15. Para a função `hexa` podemos escolher como característica a dimensão do número `x`. Como não temos

informação sobre o comportamento fora do domínio, não identificamos características para este caso.

Quanto a blocos para a característica identificada poderemos escolher $x = 0$, $0 < x < 15$ e $x = 15$, pensando nos casos fronteira (0 e 15) e num outro caso qualquer (entre 1 e 14). No entanto sabemos haver uma descontinuidade na passagem do 9 para o 10, descontinuidade essa potencialmente geradora de falhas. Podemos enriquecer a partição da caraterística para tomar em consideração este facto. Ficamos então com os blocos $x = 0$, $0 < x < 9$, $x = 9$, $x = 10$, $10 < x < 15$ e $x = 15$, que ditam os testes abaixo.

```
def hexa(x):
    """A representação hexadecimal de um número

    Args:
        x (integer): O número a convert para hexadecimal
    Pre:
        0 <= x < 15
    Returns:
        str: A representação hexa

    >>> hexa(0)
    '0'
    >>> hexa(7)
    '7'
    >>> hexa(9)
    '9'
    >>> hexa(10)
    'A'
    >>> hexa(13)
    'D'
    >>> hexa(15)
    'F'
    """
```

### Combinação de características e combinações inviáveis

```
def par(x):
    """Um dado número é par?

    Args:
        x (number): O número

    Returns:
        bool: True se x é par; False caso contrário
    """
```

Aqui identificamos facilmente uma característica baseada na *interface*: "`x` é zero". Uma outra característica importante advém da análise da *funcionalidade* da função: o facto da função devolver `True` ou `False`. As características e blocos em que estamos interessados são as seguintes:

| Característica | Blocos |
|---|---|
| `x` é zero | $z_1$ =`True`, $z_1$ =`False` |
| A função devolve `True` | $f_1$ =`True`, $f_2$ =`False` |

Claramente a combinação $z_1$ com $f_2$ é inviável, pelo que identificamos três testes ao invés dos quatro esperados.

```
def par(x):
    """Um dado número é par?

    Args:
        x (number): O número

    Returns:
        bool: True se x é par; False caso contrário


    >>> par (0)
    True
    >>> par (33)
    False
    >>> par (126)
    True
    """
```

### Mais combinações de características

```
def fusao (l1, l2):
    """Funde duas listas numa só

    Pre:
        Ambas as listas estão ordenadas
    Post:
        A lista resultante está ordenada
    Args:
        l1 (list): Uma lista
        l2 (list): A outra lista
    Returns:
        list: Uma lista ordenada contendo exactamente os
            elementos nas duas listas parâmetro. A
            multiplicidade dos elementos repetidos é
```

```
        mantida, de modo que, se a l1 tiver n
        ocorrencias do elemento x e a l2 m, o
        resultado tem n+m ocorrencias de x.
    """
```

Olhando para a interface escolhemos as características do costume: "a primeira lista está vazia" e "a segunda lista está vazia". Pensando em termos da relação entre as duas listas podemos identificar muitas outras características. Eis duas possíveis: "número de elementos em comum nas listas", "relação entre os máximos e os mínimos das duas listas". Desta análise saem as seguintes blocos.

| Característica | Blocos |
|---|---|
| `lista1` é vazia | True, False |
| `lista2` é vazia | True, False |
| Número de elementos em comum | 0, 1, $> 1$ |
| Relação min/max | $r_1 = \max(l_1) < \min(l_2)$, |
| | $r_2 = \min(l_2) < \max(l_1) < \max(l_2)$, |
| | $r_3 = \max(l_2) < \min(l_1)$, |
| | Outro |

Uma chamada de atenção para a bloco "Outro". Os blocos devem *particionar* o espaço da característica associada, isto é, não se devem sobrepor, todas juntas devem cobrir o espaço completo e nenhum bloco deverá ser vazio. As três primeiras blocos não cobrem o espaço: por exemplo, fica de fora o caso em que $\min(l_1) < \max(l_2) < \max(l_1)$, bem como o caso em que o mínimo e o máximo não estão definidos (por a lista estar vazia). O bloco "Outro" cobre todos estes casos.

Pelo critério de cobertura que utilizamos—cobertura de todas as combinações—estaríamos à espera de $2 \times 2 \times 3 \times 4 = 48$ testes, um número realmente grande. Acontece que grande parte das combinações não são viáveis. Por exemplo, não conseguimos construir um teste em que `lista1` é vazia e tem um elemento em comum com a `lista2`.

Para simplificar a construção da tabela que associa blocos a testes (e que identifica as combinações de blocos inviáveis), começamos por pensar nas listas vazias. Quando ambas as listas são vazias, que outros blocos posso utilizar? Claramente apenas o bloco que diz que o número de elementos em comum é zero e que a relação min/max é "Outro". Todas as outras combinações são inviáveis. E quando apenas a primeira lista é vazia? Neste caso só podemos contar com 0 elementos em comum e "outros" no min/max. O raciocínio é análogo para o caso em que a segunda lista é vazia. Este raciocínio leva-nos a eliminar um grande número de casos, restando apenas os $3 \times 4 = 12$ casos em que ambas as listas não são vazias. Eis uma possível tabela de testes,

| Características e blocos | | | | Testes | |
|---|---|---|---|---|---|
| Vazia1 | Vazia2 | Com. | min/max | Entrada | Resultado |
| True | True | 0 | Outro | `([],[])` | `[]` |
| True | False | 0 | Outro | `([2,3,3],[])` | `[2,3,3]` |
| False | True | 0 | Outro | `([],[7,24])` | `[7,24]` |
| False | False | 0 | $r_1$ | `([3,7],[9,17])` | `[3,7,9,17]` |
| False | False | 0 | $r_2$ | `([2,4],[3,9])` | `[2,3,4,9]` |
| False | False | 0 | $r_3$ | `([2,37,37],[1])` | `[1,2,37,37]` |
| False | False | 0 | Outro | `([3,9],[2,4])` | `[2,3,4,9]` |
| False | False | 1 | $r_1$ | Inviável | |
| False | False | 1 | $r_2$ | `([4],[2,4,8])` | `[2,4,4,8]` |
| False | False | 1 | $r_3$ | Inviável | |
| False | False | 1 | Outro | `([2,4,8],[4])` | `[2,4,4,8]` |
| False | False | >1 | $r_1$ | Inviável | |
| False | False | >1 | $r_2$ | `([1,4],[1,4,8])` | `[1,1,4,4,8]` |
| False | False | >1 | $r_3$ | Inviável | |
| False | False | >1 | Outro | `([1,4,8],[1,4])` | `[1,1,4,4,8]` |

e os testes `doctest` correspondentes:

```
def fusao (l1, l2):
    """Funde duas listas numa só

    Pre:
        Ambas as listas estão ordenadas
    Post:
        A lista resultante está ordenada
    Args:
        l1 (list): Uma lista
        l2 (list): A outra lista
    Returns:
        list: Uma lista com os elementos das duas listas
            parâmetro

    >>> fusao ([], [])
    []
    >>> fusao ([2, 3, 3], [])
    [2, 3, 3]
    >>> fusao ([],[7, 24])
    [7, 24]
    >>> fusao ([3, 7], [9, 17])
    [3, 7, 9, 17]
    >>> fusao ([2, 4], [3, 9])
    [2, 3, 4, 9]
    >>> fusao ([2, 37, 37], [1])
    [1, 2, 37, 37]
    >>> fusao ([3, 9], [2, 4])
```

```
[2, 3, 4, 9]
>>> fusao ([4], [2, 4, 8])
[2, 4, 4, 8]
>>> fusao ([2, 4, 8], [4])
[2, 4, 4, 8]
>>> fusao ([1, 4], [1, 4, 8])
[1, 1, 4, 4, 8]
>>> fusao ([1, 4, 8], [1, 4])
[1, 1, 4, 4, 8]
"""
```

# Referências

[1] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, 2008.

[2] Partition of a set. Wikipedia. https://en.wikipedia.org/wiki/Partition_of_a_set