Trabalho Computacional
Matemática Experimental

Instituto Superior Técnico

Dezembro de 2020

Clara Pereira - 99405
Diana Gaspar - 99407
Madalena Preto - 99423
Samuel Pearson - 99441
Vasco Reigadas - 99447

Índice

1 Exercício 1

Neste problema, pretende-se verificar algumas propriedades dos números inteiros, relacionados com o facto de um número ser ou não pseudoprimo numa certa base e, além disso, pseudoprimo forte nessa base.

Deste modo, recorre-se a noções como o Pequeno Teorema de Fermat, que nos diz que, se p é um primo e a um número natural, então $a^p \equiv a \pmod{p}$. Ora, daí segue-se o seguinte corolário $a^{p-1} \equiv 1 \pmod{p}$. Os pseudoprimos são números que passam este teste sem serem, de facto, primos, ou seja, $a^{n-1} \equiv 1 \pmod{n}$, com n um inteiro composto.

Para além disto, é-nos fornecido esse n, tal que $n = 2^\alpha t + 1$, candidato para ser primo, que tenha passado o teste de primalidade na base b, ou seja, é um pseudoprimo de Fermat.

1.1 Alínea a)

Nesta alínea, é necessário verificar a seguinte expressão:

$$b^{n-1} - 1 = (b^t - 1)(b^t + 1)(b^{2t} + 1)(b^{4t} + 1)...(b^{2^{\alpha-1}t} + 1) \tag{1}$$

Abordemos primeiro o problema de uma perspetiva teórica.

Se pensarmos no lado direito da equação, começamos por encontrar o produto de conjugados $(b^t - 1)(b^t + 1) = (b^{2t} + 1)$. Depois, ficamos com o seguinte produto de conjugados: $(b^{2t} + 1)(b^{2t} - 1) = (b^{4t} - 1)$. E continuamos neste ciclo em que obtemos, através de cada produto, uma nova parcela constituída por uma subtração em que duplica o coeficiente do t.

Isto possibilita a combinação da mesma com o fator seguinte, que é sempre o seu conjugado, uma soma cujo expoente também é o dobro do anterior, visto o coeficiente do t varia na expressão aumentando uma unidade o valor do expoente do 2 no expoente de b.

Eventualmente, obtemos a expressão $(b^{2^{\alpha-1}t} - 1)(b^{2^{\alpha-1}t} + 1) = (b^{2^\alpha t} - 1)$. Como $n = 2^\alpha t + 1$, chegamos à conclusão que $(b^{2^\alpha t} - 1) = b^{n-1} - 1$, completando assim a implicação da direita para a esquerda.

Do mesmo modo, considerando o primeiro lado da equação, poderíamos dizer que este era uma diferença de quadrados e desdobrá-lo sequencialmente, recorrendo a um pensamento semelhante para obter a implicação da esquerda para direita, completando, assim, a equivalência.

Ora, o *Mathematica* contribui para esta demonstração, confirmando sequencialmente a combinação de fatores do lado direito da equação, até se obter algo equivalente ao primeiro $(b^{2^\alpha t} - 1)$.

Para além disso, definimos um programa que verifica a validade da equação para uma dada base, um a e um t e a um teste de pseudoprimalidade de um número baseado do corolário do Pequeno Teorema de Fermat descrito acima, *pseudoprimo*. A partir destes, definimos uma função que verifica simultâneamente ser pseudoprimo numa certa base e satisfazer a equação. Experimentámos alguns casos.

1.2 Alínea b)

Neste exercício, é-nos dado o pseudocódigo necessário para definir um teste que indica se um certo número é pseudoprimo forte na base b, ou seja, se divide um e só

um dos fatores do lado direito da equação (1), situada na alínea a).

Deste modo, implementou-se o mesmo pseudogódigo através da linguagem *Mathematica* necessária, criando o critério pseudopforte, que será utilizado nas seguintes alíneas deste problema. Incluem-se também alguns exemplos do seu funcionamento.

1.3 Alínea c)

A alínea c) pede várias informações acerca dos pseudoprimos na base 2: o número de pseudoprimos até 10^6 e até 10^9, o número de pseudoprimos fortes entre esses, e o primeiro pseudoprimo. Recorremos assim, a um teste de pseudoprimalidade de um número baseado do corolário do Pequeno Teorema de Fermat descrito acima, *pseudoprimo*, definido na alínea a), e ao teste descrito na alínea b), que testa a pseudoprimalidade forte numa certa base, *pseudopforte*.

De forma a tornar o teste de pseudoprimalidade mais eficientemente e uma vez que apenas nos interessam os números compostos com estas propriedades, verificou-se que era mais rápido verificar a pseudoprimalidade para uma certa passa numa lista de números compostos até n do que numa lista de primos e compostos até n: no caso da base 2 até 10^6, a primeira instância demora menos de 70 segundos, enquanto a segunda demora 75. Apesar desta diferença parecer pouca ao início, será muito mais significativa se considerarmos listas maiores como 10^9. Deste modo, vamos sempre obter o número de pseudoprimos compostos, ignorando os primos.

Em primeiro lugar, determinámos uma lista de compostos nesta base até 10^6. Daqui, conseguimos logo determinar o primeiro pseudoprimo em base 2, que corresponde ao primeiro elemento da lista, 341, através do comando *Part*. O número de pseudoprimos, 245, é dado pelo comprimento da lista, ou seja, pelo comando *Length*. Seguidamente, testámos cada elemento desta lista de pseudoprimos para a pseudoprimalidade forte, uma vez que nenhum número pode ser pseudoprimo forte numa certa base sem ser pseudoprimo nessa base. Então, o comprimento desta lista também dos dá o número de pseudoprimos fortes, 46.

Seguiu-se o mesmo procedimento para obter o número de pseudoprimos e de pseudoprimos fortes na base 2 até 10^7, o que demorou cerca de duas horas e meia, obtendo 750 pseudoprimos, dos quais 162 são fortes. Não se fez para 10^9 uma vez que demoraria, estimamos, mais de cem vezes esse tempo. Seria necessário otimizar o teste de pseudoprimalidade para obter os resultados pedidos. Contudo, sabemos, após pesquisar, que o resultado seria 5597 compostos pseudoprimos.

1.4 Alínea d)

Nesta alínea são pedidos mais testes concretos de pseudoprimalidade, em particular, o número de números que satisfazem, simultaneamente, a pseudoprimalidade nas bases 2, 3, 5 e 7 até $25 * 10^9$, e, entre esses, é pedido para que seja encontrado o único pseudoprimo forte em todas estas bases.

Uma vez que se provou extremamente difícil calcular a lista de pseudoprimos para 10^9, utilizámos estas listas já calculadas para outros testes de primalidades para 10^7, em vez de testar para $25*10^9$. Isto afeta, obviamente, os nossos resultados, alterando tanto o número de pseudoprimos nas bases 2, 3, 5 e 7, tanto a interseção dos pseudoprimos fortes nestas bases.

Deste modo, foram-se calculando listas sucessivas através do comando *Select* e do

teste *pseudoprimo*: primeiramente, dos pseudoprimos em base 2; de entre esses, os pseudoprimos em base 3; depois 5; e, por fim, 7. Assim, são cada vez menores as listas que temos de testar, contribuindo para a eficiência da resolução. Há, portanto, 63 pseudoprimos nestas bases, nestas condições.

Por fim, encontrar o único pseudoprimo forte em todas estas bases não estava, logo de partida, garantindo, uma vez que estamos a testar uma parte pequena dos números que nos eram pedidos, sendo muito mais provável que o tal número que satisfaz todos os testes ser maior que 10^7. Fomos calculando, a partir da lista de pseudoprimos obtidos (de forma a maximizar a eficiência), os pseudoprimos fortes: primeiro em base 2; de entre esses, em base 3; e assim a diante. Contudo, obtivemos uma lista vazia, isto é, o pseudoprimo forte nas quatro bases seria de uma ordem superior (estimamos que seria, possivelmente, superior a 10^9).

Após pesquisar, apesar de não termos conseguido confirmar estes resultados experimentalmente, descobrimos que há 1770 pseudoprimos nas quatro bases e que a nossa estimativa estava correta: há um número composto pseudoprimo forte nas quatro bases é 3215031751, superior a 10^9. Conseguimos verificar a validade a última afirmação com o teste *pseudopforte*.

1.5 Alínea e)

Esta alínea requer que calculemos os pseudoprimos na base 2 entre 10^{100} (1 *googol*) e $10^{100}+1000$. Desta forma, apresenta-nos um novo problema, o *overflow* no programa *Mathematica*, ou seja, a impossibilidade de escrever algumas soluções uma vez que os números têm demasiados algarismos. Deste modo, é-nos pedido que o nosso código permita que este teste corra, devolvendo os pseudoprimos na forma $n = 10^{100} + x, 0 \leq x \leq 1000$.

Efetivamente, resolvemos o problema em duas fases: tendo em conta uma lista de 0 a 1000, aplicámos o teste pseudoprimo a cada um dos seus elementos somando-lhes um *googol*, e, de seguida, traduzimos esse resultado, quando necessário, noutra lista, de forma a apresentar os pseudoprimos na forma acima descrita, recorrendo ao comando *HoldForm*.

Ao aplicarmos o teste *pseudoprimo*, obtivemos uma lista vazia. Todavia, ao aplicarmos o teste de pseudoprimalidade forte na mesma, *pseudopforte*, obtemos, curiosamente, dois possíveis elementos: $10^{100} + 267$ e $10^{100} + 949$. Isto tem a ver com o facto de o primeiro programa envolver utilizar o número que quer testar como expoente na base 2, levando ao tal *overflow*, enquanto o segundo segue um outro caminho que não envolve esta potenciação, evitando o *overflow*.

Contudo, o teste *pseudopforte* não exclui primos e devemos notar que ambas as soluções obtidas são números primos. Portanto, podemos concluir que não há pseudoprimos fortes em base 2 compostos no intervalo considerado e não conseguimos tirar conclusões concretas acerca dos pseudoprimos não fortes em base 2 no mesmo.

1.6 Alínea f)

Nesta etapa, introduzimos o conceito de números de Carmichael neste problema: um número composto n, ímpar, que passa o teste de pseudoprimalidade na base b para cada b, $1 < b < n$. Ou seja, um número de Carmichael é um pseudoprimo em todas as bases entre 2 e ele próprio, não inclusive. As bases devem ser coprimas com o próprio

número.

Deste modo, definimos o teste *carmichael*, que cria uma lista de valores lógicos (*True* ou *False*) de pseudoprimalidade entre um número e todas as suas bases e, seguidamente, verifica se se obtiveram apenas valores *True*. Ou seja, se este for o caso, é porque ele é pseudoprimo para todas as bases, isto é, é, de facto, um número de Carmichael.

Recorremos, portanto, ao programa definido anteriormente, *pseudoprimo*. Uma vez que para números cada vez maiores há mais bases a verificar, este programa torna-se cada vez mais ineficiente e demorado. Assim, selecionámos todos os números que satisfazem o teste *carmichael* até 10 000, o que ainda se provou eficiente.

Como apenas testamos números compostos n, sabemos que $n = p_1^{e_1} p_2^{e_2} ... p_k^{e_k}$, logo faz sentido calcularmos o número de fatores primos distintos de cada número n. Assim, através do programa próprio do *Mathematica* denominado *PrimeNu*, que calcula o número de fatores primos distintos de um dado número, podemos selecionar da lista dos números de Carmichael obtida aqueles compostos por quatro fatores primos distintos. Ao obtermos uma lista vazia, chegámos à conclusão que o primeiro número de Carmichael com quatro fatores primos diferentes é maior que 10 000.

Ao pesquisarmos, confirmámos que o primeiro número de Carmichael que satisfaz estas condições é 41041, o que verificámos experiencialmente.

1.7 Alínea g)

Nesta alínea, fomos desafiados a provar que o número composto n é um número de Carmichael se e só se não for divisível por um quadrado perfeito (diferente de 1) e todos os seus divisores primos p são tais que $p-1|n-1$. Esta ideia é conhecida como o Teorema de Korselt.

Experimentalmente, criámos outro teste *carmichael2*, que verifica se um número é um número de Carmichael de acordo com a definição de Korselt, recorrendo à lista dos divisores do número e verificando várias propriedades dos mesmos. Ao testar para todos os números até 10 000, chegámos às mesmas conclusões que na alínea anterior. Este é um indicativo de que ambos os testes são determinam os mesmos números de Carmichael. Contudo, para complementar a nossa confirmação, testámos ambos para uma lista ainda maior, até 50 000. Obtivémos novamente, como seria de esperar, resultados iguais.

Para além da componente experimental, também podemos verificar a igualdade entre os dois testes teoricamente, seguindo o pensamento de Korselt na prova do seu teorema.

Temos dois casos possíveis: se o p é par, pelo Pequeno Teorema de Fermat, temos que $-1 \equiv 1 \pmod{p}$, logo $n = 2$. Temos também que $n = p_1^{e_1} p_2^{e_2} ... p_k^{e_k}$, com p_k fatores primos distintos ímpares elevados a expoentes $e_k \geq 1$, um problema abordado superficialmente na alínea anterior. Para cada $1 \leq i \leq k$, escolhemos uma raiz primitiva $a_i \pmod{p_i^{e_i}}$.

Pelo Teorema Chinês dos Restos, encontramos o a inteiro tal que $a \equiv a_i \pmod{p_i^{e_i}} \Rightarrow a^{n-1} \equiv 1 \pmod{n}$. Como a é a raiz primitiva mod $p_i^{e_i}$, segue-se logicamente que $ord_{p_i^{e_i}}(a_i) = \phi(p_i^{e_i}) = p_i^{e_i - 1}(p_i - 1)$ divide $n-1$. Mas $mdc(p_i, n-1) = 1$ e $p_i - 1 | n-1$.

O mesmo pensamento poderia ser feito na ordem inversa, garantindo a implicação para os dois lados.

Portanto, confirmámos de duas formas, experimental e teórica que um número

composto n é um número de Carmichael se e só se não for divisível por um quadrado perfeito (diferente de 1) e todos os seus divisores primos p são tais que $p-1|n-1$.

Por fim, podemos também concluir que o programa *carmichael* é menos mais eficiente do que *carmichael2*, uma vez que no primeiro temos de verificar a pseudoprimalidade de um número com todas as bases possíveis, o que é um processo longo (demorou mais de 20 minutos a correr para uma lista de 50 000 números), enquanto o segundo apenas acede aos divisores do número em causa e verifica certas propriedades dos mesmos (demorando menos de 2 segundos para a mesma lista, o que é uma diferença significativa).

Ou seja, o Teorema de Korselt é uma boa solução para a ineficiência do teste *carmichael*, sendo mais rápido pôr em prática este teorema que a definição de número de Carmichael considerada na alínea anterior.

2 Exercício 2

Ao longo do exercício 2, várias alíneas envolvem determinar se certos b são resíduos quadráticos módulo p, por isso, começou-se por escrever o programa *residuoQ* que, dado um inteiro b e um número primo p (primos entre si), calcula o resultado, através do critério de Euler, e devolve *True* se b é resíduo quadrático módulo p, e *False* se é resíduo não quadrático módulo p.

2.1 Alínea a)

O programa *afirmacao* recebe os inteiros c e d, cujo produto é b, e o número primo p, e através do comando *Equivalent*, compara o valor lógico da afirmação "Se b for composto então $b = cd$ é resíduo quadrático módulo p" com a afirmação "ou c e d são ambos resíduos quadráticos ou nenhum dos dois é resíduos quadráticos módulo p", utilizando o programa *residuoQ* criado anteriormente.

De seguida, para verificar se a afirmação é verdadeira, usa-se *Table* para gerar cópias de afirmacao, com p a tomar um valor nos quatro primeiros primos, e com c e d a variar entre 1 e $p-1$. c e d nunca podem ter o mesmo valor que p, pois não é possível calcular *residuoQ*[c,p] ou *residuoQ*[d,p] se estes tiverem o mesmo valor (o máximo divisor comum seria diferente de 1).

Dispondo as cópias numa tabela, observa-se que a afirmação tem sempre o valor *True* para todos os valores testados.

Esta afirmação pode ser também demonstrada teoricamente: o número composto b ser resíduo quadrático módulo p equivale a dizer que $b^{\frac{p-1}{2}} \equiv 1 \pmod{p} \Rightarrow c^{\frac{p-1}{2}} d^{\frac{p-1}{2}} \equiv 1 \pmod{p}$. Logicamente, para satisfazer esta equação modular, $c^{(p-1)/2}$ e $d^{(p-1)/2}$ ou são ambos 1 (mod p) ou -1 (mod p). Conclui-se então que a afirmação é verdadeira.

2.2 Alínea b)

Dados um primo b e um número inteiro *nmax* (maior ou igual a 3), o programa *primosnmax* usa a ferramenta *Select* para criar uma lista de primos ímpares inferiores a *nmax*, e depois novamente para selecionar os valores dessa lista para os quais b é resíduo quadrático.

Posteriormente, apresentam-se numa tabela os valores de b pedidos no enunciado, e a lista correspondente.

2.3 Alínea c)

O código *reciprq* tem como objetivo, dados os dois primos $p,q \geq 3$, verificar, em primeiro lugar, se p e q são ambos congruentes com 3 módulo 4, com o comando *If*.

Caso não sejam, *reciprq* determina se "p é resíduo quadrático módulo q se e só se q é resíduo quadrático módulo p", com o *Equivalent*.

Caso ambos sejam congruentes com 3 módulo 4, *reciprq* vai determinar a veracidade da afirmação "p é resíduo quadrático módulo q se e só se q é resíduo quadrático módulo p".

Para testar experimentalmente o resultado, criaram-se com o *Subsets* conjuntos de dois primos menores que 20000 (que correspondem a p e q), e aplicou-se *reciprq*, verificando-se, tal como pretendido, que o valor obtido é *True* para todos.

2.4 Alínea d)

A primeira parte da Lei da Reciprocidade Quadrática afirma que, dados primos ímpares p e q, p é resíduo quadrático módulo q se e só se q é resíduo quadrático módulo p, se pelo menos um dos dois primos é congruente com 1 módulo 4.

No ponto **ii.**, 5 é congruente com 1 módulo 4, logo 5 é resíduo quadrático módulo p se e só se p é resíduo quadrático módulo 5. Os resíduos quadráticos de 5 são ± 1, logo, prova-se o ponto **ii.**. Para o ponto **iv.**, o raciocínio é semelhante, mas temos 13 em vez de 5, e os resíduos quadráticos de 13 são $\pm 1, \pm 3, \pm 9$.

A segunda parte da Lei da Reciprocidade Quadrática afirma que, dados esses mesmos primos, se p e q são ambos congruentes com 3 módulo 4, então p é resíduo quadrático módulo q se e só se q é resíduo não quadrático módulo p.

Nos pontos **i.** e **iii.**, 3 e 7 são congruentes com 3 módulo 4, logo 3 e 7 são resíduos quadráticos módulo p se e só se p é resíduo não quadrático módulo 3 (no caso do 3) ou 7 (no caso do 7). Como em **i.**, $p = 12$ e 12 é congruente com 0 módulo 3 e, no **iii.**, $p = 28$ e 28 é congruente com 0 módulo 7, p claramente não é resíduo quadrado em nenhum dos casos e ficam ambas as alíneas provados.

Os resíduos quadráticos de 5 são ± 1, logo, prova-se o ponto **ii.**. Para o ponto **iv.**, o raciocínio é semelhante, mas temos 13 em vez de 5, e os resíduos quadráticos de 13 são $\pm 1, \pm 3, \pm 9$, logo também fica provado.

Nesta alínea, para cada um dos pontos **i.,ii.,iii.** e **iv.**, foram criadas as funções, respetivamente, *confirmari, confirmar ii, confirmariii* e *confirmariv*, que recebem um primo ≥ 3, e verificam as afirmações do enunciado de cada uma, com a ferramenta *Equivalent*. De seguida, com *Apply*, testa-se para valores de p até 1000000.

`Nota:` no ponto **i.** o *Range* começa em 4, pois não é possível determinar para o valor 3 com *residuoQ*; no ponto **ii.**, a ferramenta *Drop* retira *!residuoQ*[5, 5] da lista que se obtém com o *Map* (à semelhança dos pontos **iii.** e **iv.**).

2.5 Alínea e)

Em primeiro lugar, definiu-se a função *fermat*, que recebe um inteiro $k \geq 0$ e devolve o número de Fermat F_k.

De seguida, programou-se *fermatresiduo*, que recebe também $k \geq 0$ e, se F_k for primo, verifica se 3 é resíduo não quadrático módulo esse F_k.

Com o auxílio da ferramenta *Map*, testou-se *fermatresiduo* até $k = 16$ que devolveu *False* para todos os k cujo F_k é primo (não devolve nada se F_k for composto), o que significa que 3 é resíduo não quadrático módulo esse F_k.

As classes de congruência de 2^k em módulo 12 são $\{1, 2, 4, 8\}$, logo as de 2^{2^k} são $\{2, 4, 8\}$. Logo as classes de congruência de F_k (mod 12) são $\{3, 5, 9\}$. Quando F_k é 3 ou 9 (mod 12), não é primo, logo, quando F_k é primo, $F_k \equiv 5 \bmod 12$. Se 3 fosse resíduo quadrático módulo F_k primo, $3^{(3-1)/2} \equiv 1 \pmod{F_k} \Rightarrow 3 \equiv 1 \pmod{F_k}$. Mas, como $F_k \equiv 5 \pmod{12}$, esta equação modular é impossível. Podemos concluir que 3 é resíduo não quadrático de F_k primo.

2.6 Alínea f)

A alínea f) tem como objetivo confirmar o teste de Pépin. Para isso, criou-se o programa *pepin*, que recebe um $k \geq 0$ e, à semelhança da alínea a), usa o comando *Equivalent* para comparar o valor lógico das duas proposições "o número de Fermat F_k é primo" e "$3^{(F_k-1)/2} \equiv -1 \pmod{F_k}$", utilizando também a função *fermat* definida na alínea anterior.

A seguir, usa-se *Map* e *Apply* para verificar o teste de Pépin para *Range*[16] e, tal como pretendido, o resultado é *True*.

Assumindo que $3^{(F_k-1)/2} \equiv -1 \bmod F_k$, então temos que $3^{F_k-1} \equiv 1 \bmod F_k$, e, como $F_k - 1 = 2^{2^k}, 3^{2^{2^k}} \equiv 1 \bmod F_k$. Assumindo que F_k é primo, pelo critério de Euler, temos que: $3^{(F_k-1)/2} \equiv 3/F_k \pmod{F_k}$, sendo $3/F_k$ o símbolo de Legendre, ou seja, poderá ser 1, -1 ou 0. Isto implica que $2^{2^k} \equiv 1 \pmod 3$, logo $F_k \equiv 2 \pmod 3$ e $F_k/3 \equiv 1 \pmod 3$. Como $F_k \equiv 1 \pmod 4$, $3/F_k \equiv -1 \pmod 4$, pela Lei da Reciprocidade Quadrática. Ao mesmo tempo, sabemos que 3 é resíduo não quadrático módulo número de Fermat Primo. Portanto fica provado que $3^{(F_k-1)/2} \equiv -1 \pmod{F_k}$ se e só se F_k é primo.

2.7 Alínea g)

O programa *eqmod* recebe os inteiros a, b, c, e o primo p e verifica primeiramente se ocorre o caso **i.**, ou seja, se p é igual a 2. Nesse caso, aplica *Reduce* a $(a+b)x+c == 0$ pois, se $x^2 \equiv x \pmod 2$, temos $ax^2 + bx + c \equiv 0 \pmod 2 \Leftrightarrow ax + bx + c \equiv 0 \pmod 2 \Leftrightarrow (a + b)x + c \equiv 0 \pmod 2$.

Se o primeiro caso não se verificar, o programa *eqmod* vai testar se ocorre o caso **ii.**, ou seja, se a é divisível por p. Se for *True*, aplica *Reduce* a $bx + c == 0$ pois, se $p|a$, então $a \equiv 0 \pmod p$ e temos que $bx + c \equiv 0 \pmod{\text{p}}$.

Se este caso também não se verificar, o caso **iii.** tem de ocorrer necessariamente, e aplica-se *Reduce* à equação $(x + 2^{-1}a^{-1}b)^2 \equiv 4^{-1}a^{-2}b^2 - a^{-1}c$.

Por fim, testou-se o programa *eqmod* para diferentes valores de a, b, c e p. O programa devolve *False* quando a equação não tem soluções.

3 Exercício 3

3.1 Alínea a)

Para representar este grafo, é útil definir uma função que, dados pares de inteiros m e n devolve as arestas orientadas $m \to n$ se e só se mn é primo. Note-se que, em qualquer intervalo escolhido, para $m \neq 0$, mn ser primo implica que n é ímpar.

A função *Compatíveis* recebe duas listas (a primeira de possíveis m e a segunda de possíveis n) e devolve uma lista com as arestas orientadas que verificam a condição acima. Neste caso, $80 \leq m \leq 95$ e $81 \leq n \leq 95$ (sabendo que é ímpar). Esta lista de arestas define o grafo *g*. Aplicando o comando *GraphPlot*, e alguns comandos relativos à apresentação, obtemos o grafo acima.

3.2 Alínea b)

Com o comando *Manipulate*, podemos observar as variações possíveis do grafo inicial, quando variamos m e n em função do intervalo $[10j,\ 10j + 15]$, com $j = 1$, 2, ..., 10. Basta colocar como parâmetros da função *Compatíveis* a lista de todos os inteiros pertencentes ao intervalo $[10j,\ 10j + 15]$ e a lista de todos os inteiros ímpar compreendidos nesse mesmo intervalo , ou seja, *Range*[10j, 10j + 15] e *Range*[10j, 10j + 15,2], respetivamente.

4 Exercício 4

Para efeitos de otimização, na função *first4* verificamos se i é divisível por 6 e por 2, porque se for divisível por 6 é semiperfeito, logo, não pode ser estranho. E sabe-se que se um número for ímpar, ele não é estranho (se estivermos a falar de números de ordens inferiores a 10^{21}, o que é certamente o caso, visto que procuramos obter apenas os primeiros 4 números estranhos).

Para melhor eficiência ao verificar se a soma de algum subconjunto dos divisores de n é igual a n, implementou-se um algoritmo recursivo.

A ideia básica do algoritmo é que, por exemplo, para chegar a uma soma igual a 70 a partir dos divisores próprios de 70, que são $\{1, 2, 5, 7, 10, 14, 35\}$, o 35 tem de estar incluído, porque uma soma de todos os outros divisores, exceto o 35, seria insuficiente ($1+2+5+7+10+14 = 39 < 70$). O problema é então reduzido a determinar se existe uma soma de algum subconjunto de $\{1, 2, 5, 7, 10, 14\}$ que seja igual a $70 - 35 = 35$.

Podemos repetir o mesmo processo usado anteriormente, aplicando esta lógica recursivamente até acontecer uma de duas coisas: ou chegamos a 0, caso em que o número não é estranho, ou esgotam-se os divisores próprios sem que isso aconteça, e portanto o número será estranho.

Seja p um número primo maior ou igual a 149. Os divisores próprios de $70p$ são os divisores de 70, mais o próprio p multiplicado por cada divisor próprio de 70 (uma vez que p só tem como divisores 1 e o próprio p).

Seja $w(x)$ o conjunto das somas de cada subconjunto dos divisores próprios de x. Como p é primo, para poder existir u pertencente a $w(70p)$ tal que $u = 70p$, tem de existir q pertencente a $w(70)$ tal que $q = w(70)$, pois a soma dos divisores de $70 = 144 < 149 \leq p$. Vejamos o que se segue:

Tendo em conta que os divisores próprios de $70p$ são os divisores de 70, mais o próprio p multiplicado por cada divisor próprio de 70, para existir u pertencente a $w(70p)$ tal que $u = 70p$, teria de acontecer uma das duas opções seguintes:

- Existir uma soma de algum subconjunto dos divisores próprios de 70 igual a 70 (pois cada elemento desse subconjunto multiplicado por p seria também um divisor próprio de $70p$ e portanto teríamos $w(u) = 70p$), o que não acontece porque 70 é número estranho.

- Haver uma soma, S, de uma soma de divisores próprios de 70 multiplicada por p (que teria de ser inferior a $70p$) com outra soma de divisores de 70 igual a um múltiplo de p, tal que $S = 70p$. Ora, como a soma de todos os divisores de 70 é 144, e $p \geq 149 > 144$, isso também é impossível.

Logo, não existe subconjunto de divisores próprios de $70p$ tal que a sua soma seja igual a $70p$, para todo p primo maior ou igual a 149.

Conclui-se então que todos os números da forma $70p$, onde $p \geq 149$ é primo, são números estranhos.

5 Exercício 5

Para a resolução deste exercício definiu-se a função *fatorizacoes*, que dado um inteiro positivo n devolve a lista de pares $\{x, y\}$ (x e y naturais), cada um com metade do número de dígitos de n e não ambos a terminar em zero, tais que os dígitos de x e y são precisamente os dígitos de n. Ou seja, a lista dos dígitos de x e de y é igual à lista de dígitos de n, a menos de permutação dos seus elementos.

Dentro da função temos a variável *potenciais*, que corresponde à lista dos divisores de n com metade dos seus dígitos. São então verificadas todas as condições necessárias que devem ser satisfeitas e, se o forem, será adicionado à lista um par $\{x, y\}$. Quando a lista deixa de ser vazia significa que o número n satisfaz as condições do enunciado.

Procedeu-se então à determinação dos números com 2 e 4 dígitos desta forma, tendo-se chegado à conclusão que não existem números destes com 2 dígitos e que os números com 4 dígitos são: $\{1260, 1395, 1435, 1530, 1827, 2187, 6880\}$.

Verificou-se que existem 148 números deste tipo com 6 dígitos e pelo menos 701 números deste tipo com 8 dígitos (sendo que o programa demora muito tempo a correr e quando se abortou a avaliação tinham-se contado 701 números).

6 Exercício 6

6.1 Alínea a)

A igualdade em questão pode ser provada recorrendo ao método de indução. Provar $P(0)$ é trivial, pois $r^0 + q^0 + s^0 = 1 + 1 + 1 = 3 = P(0)$. Da fórmula de recorrência e da hipótese de indução ocorre que

$$\begin{gathered} P(n) = P(n-2) + P(n-3) \\ \Rightarrow P(n) = r^{n-2} + q^{n-2} + s^{n-2} + r^{n-3} + q^{n-3} + s^{n-3} \\ \Rightarrow P(n) = r^{n-3}(r+1) + q^{n-3}(q+1) + s^{n-3}(1+s) \\ \Rightarrow P(n) = r^{n-3}(r^3) + q^{n-3}(q^3) + s^{n-3}(s^3) \Rightarrow P(n) = r^n + q^n + s^n \ \ c.q.d \end{gathered}$$

$r + 1 = r^3, q + 1 = q^3, s + 1 = s^3$ pois são raízes da equação $x^3 - x - 1 = 0$.

Assim, porque $p(n)$ implica $p(n+1)$ e $p(0)$ é verdadeira, o método de indução garante que $p(n)$ é verdadeira.

De facto, nesta alínea utilizou-se programação recursiva para definir a função Perrin. Esta recebe um inteiro positivo n e devolve o n-ésimo número de Perrin. No *notebook* encontram-se exemplos do bom funcionamento desta função, mas devido a baixa eficiência deste paradigma de programação, estes exemplos correspondem a valores bastante baixos (ordem de grandeza 10^3).

Além disto, recorrendo ao comando *Solve* obteve-se as três soluções da equação $x^3 - x - 1 = 0$. Tendo sido estes valores designados por r, s e q. Com recurso aos valores adquiridos, foi possível criar a função *Perrin1*, que recebe um inteiro positivo e devolve soma das potências de índice n de r, s e q.

Com recurso aos comandos *AllTrue* e *Equivalent*, estabeleceu-se que as funções são equivalentes para os primeiros 100 inteiros positivos, não sendo possível confirmar valores superiores devido ao motivo suprarreferido. Esta prova empírica, embora, pouco precisa, corrobora a demonstração teórica. Foi ainda usado o comando *FindInstance* para procurar inteiros positivos que tornassem as expressões diferentes não tendo sido encontrados nenhuns.

6.2 Alínea b)

Esta igualdade pode ser demonstrada com auxílio do método de indução. Para $n = 0$ a propriedade é obviamente verdadeira, pois $P(0) = 3, P(1) = 0$ e $P(2) = 2$. Suponhamos a propriedade verdadeira para n, então:

$$\begin{pmatrix} P_n \\ P_{n+1} \\ P_{n+2} \end{pmatrix} = A^n \begin{pmatrix} 3 \\ 0 \\ 2 \end{pmatrix} \Rightarrow \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix} \begin{pmatrix} P_n \\ P_{n+1} \\ P_{n+2} \end{pmatrix} = A^{n+1} \begin{pmatrix} 3 \\ 0 \\ 2 \end{pmatrix} \Rightarrow$$

$$\Rightarrow \begin{pmatrix} P_{n+1} \\ P_{n+2} \\ P_{n+1} + P_n \end{pmatrix} = A^{n+1} \begin{pmatrix} 3 \\ 0 \\ 2 \end{pmatrix} \Rightarrow \begin{pmatrix} P_{n+1} \\ P_{n+2} \\ P_{n+3} \end{pmatrix} = A^{n+1} \begin{pmatrix} 3 \\ 0 \\ 2 \end{pmatrix}$$

Como $P(0)$ é verdadeira , e a verdade da hipótese implica a verdade da tese a hipótese fica provada.

Com efeito, nesta alínea definiu-se a função *Perrin2* que funciona de forma semelhante à função *Perrin*, mas utiliza a definição matricial dos números de Perrin. Sem

dúvida esta definição é bastante mais eficiente, sendo possível calcular números de Perrin de elevada ordem(10^7), como pode ser visto nos exemplos contidos no *Notebook*. Deste modo, será a definição utilizada nos exercícios seguintes com intuito de maximizar eficiência.

De facto, utilizando uma construção semelhante à da alínea a), mostra-se que as funções são equivalentes para os primeiros 1000 naturais, complementado a prova por indução.

Com auxílio do comando *EigenValues*, foi possível estabelecer que os valores próprios da matriz A são as raízes r, q e s, estando tal na base da igualdade entre as expressões.

6.3 Alínea c)

Este exercício tem como objetivo confirmar a igualdade

$$r = \sqrt[3]{1 + \sqrt[3]{1 + \sqrt[3]{1 + ...}}} \Leftrightarrow r = \sqrt[3]{1 + r} \Leftrightarrow r^3 = 1 + r \Leftrightarrow r^3 - r - 1 = 0$$

O número plástico corresponderá então à raiz real da equação $x^3 - x - 1 = 0$ c.q.d.

De facto, utilizando o comando *FullSimplify*, foi possível calcular o valor da diferença entre r^3 e $1 + r$, obtendo-se como resultado 0. Confirmando-se assim que r é da forma apresentada.

6.4 Alínea d)

Provar que

$$\lim_{n\to\infty} P(n) = r^n$$

corresponde a provar que

$$\lim_{n\to\infty} q^n + s^n = 0.$$

Sejam $q = a + bi$ e $s = a - bi$. Utilizando o binómio de Newton é possível escrever a expressão como:

$$q^n + s^n = \sum_{k=0}^{n} \binom{n}{k} a^{n-k}(bi)^n + \sum_{k=0}^{n} \binom{n}{k} a^{n-k}(-bi)^n =$$

$$= \sum_{k=0}^{n} \binom{n}{k} a^{n-k}((bi)^n + (-bi)^n)$$

Para n ímpar:

$$(bi)^n + (-bi)^n = 0$$

$$\sum_{k}^{n} \binom{n}{k} a^{n-k}((bi)^n + (-bi)^n) = 0$$

Para n par e múltiplo de 4:

$$(bi)^n + (-bi)^n = 2b^n$$

$$\sum_{k}^{n} \binom{n}{k} a^{n-k}((bi)^n + (-bi)^n) = \sum_{k}^{n} \binom{n}{k} a^{n-k} 2b^n$$

Para n par e múltiplo de 2:

$$(bi)^n + (-bi)^n = -2b^n$$

$$\sum_k^n \binom{n}{k} a^{n-k}((bi)^n + (-bi)^n) = -\sum_k^n \binom{n}{k} a^{n-k} 2b^n$$

Embora a manipulação acima não prove o limite desejado, prova que a expressão vai oscilar entre valores positivos, zero e valores negativos, levando a crer que a expressão se anula no infinito.

Utilizando o comando *Limit*, conclui-se que $\lim_{n\to\infty} q^n + s^n = 0$, comprovando o resultado requisitado.

6.5 Alínea e)

Para a resolução deste exercício criou-se a função *DivisaoPerrin* ,que recebe um inteiro n positivo(n) e devolve *True* se n divide $P(n)$ e *False*, caso contrário. Para tal utilizou-se o comando *Mod* e a definição *Perrin2*. Criaram-se várias listas com múltiplos primos e recorrendo ao comando *AllTrue* estabelecemos que para n primo, n divide $P(n)$.

6.6 Alínea f)

Este exercício tem como objetivo obter uma lista dos 10 primeiros pseudoprimos de Perrin. Para tal criou-se a função PseudoPrimoP, que recebe um inteiro positivo(n) e devolve *True* se n for um pseudoprimo de Perrin e *False* nos restantes casos. Utilizando um ciclo *While*, estabeleceu-se um programa que se x for pseudoprimo de Perrin, adiciona x à lista de resultados.

Este ciclo inicia-se com $x = 1$ e é percorrido incrementando x em uma unidade, terminando quando a lista de resultados atinge comprimento 10. Assim, obter-se-iam os 10 primeiros pseudoprimos de Perrin, mas devido a elevada exigência computacional deste processo não foi possível. Mas com pesquisa foi possível obter os números desejados que foram testados no *Notebook* com a função *PseudoPrimoP*.

7 Exercício 7

O exercício em questão aborda o teorema de Zeckendorf que afirma que qualquer número natural pode ser representado, de forma única como a soma de elementos não consecutivos da sucessão de Fibonacci.

Tal pode ser provado recorrendo ao método indução. $P(1)$ é verdadeira pois 1 é elemento da sucessão de Fibonacci. Suponha-se a propriedade verdadeira para todos os naturais até n. Se n pertencer à sucessão de Fibonacci, então a propriedade fica provada. Caso contrário, irá existir um j tal que $F(j) < n < F(j+1)$. Seja $a = n - F(j)$, por hipótese de indução a tem uma representação de Zeckendorf.

Das desigualdades e da igualdade acima, obtém-se que $a < F(j+1) - F(j) = F(j-1)$. Deste modo, a representação de Zeckendorf de a não contém $F(j-1)$. Como tal, n pode ser escrito como a soma da representação de Zeckendorf de a e $F(j)$, tendo, deste modo, uma representação de Zeckendorf. Como $F(1) = F(2)$, obtém-se a expressão do enunciado, pois basta garantir que se considera sempre $F(2)$.

7.1 Alínea a)

Para a resolução deste exercício criou-se a função *Indices*, que recebe um inteiro positivo n e devolve a lista dos índices dos números de Fibonacci não consecutivos que somados dão n. Com efeito, recorre-se à ferramenta *NestWhile* para procurar o índice do número de Fibonacci mais próximo de m alterando o valor de m para este número.

Repete-se este processo até m ser 0. Os vários valores de m corresponderão aos números de Fibonacci e serão registados utilizando o mecanismo *Reap* e *Sow*. É de notar que, como o algoritmo procura sempre o número de Fibonacci mais próximo inferiormente de m, nunca utilizará números de Fibonacci consecutivos, pois dois números consecutivos somados resultam no número seguinte.

Por um lado, utilizando esta função, definiu-se a função *RepresentacaoZeckendorf*, que recebe um inteiro positivo n e devolve os números de Fibonacci não consecutivos cuja soma é n. Esta função corresponde a aplicar ferramenta *Fibonacci* ao resultado da função *Indices*.

Por outro lado, foi criada a função *RepresentacaoZeckendorfBinaria* que, utilizando um ciclo *For*, percorre a lista dos índices criando a representação binária através da alteração de uma lista de zeros, criada com o comando *ConstantArray*.

Por fim, utilizaram-se as duas funções suprarreferidas para criar a função *Zeckendorfinal* que retorna ambos os resultados ajustados, correspondendo à função requisitada no enunciado. Utilizou-se o comando *Reverse* para que os resultados fossem apresentados como pedidos.

7.2 Alínea b)

Neste exercício é necessário verificar que a representação de Zeckendorf de um número é única. Esta propriedade pode ser comprovada de forma teórica.

Suponha-se a existência de dois conjuntos não vazios de termos não consecutivos da sucessão de Fibonacci S e T, que têm a mesma soma. Sejam S' e T' os conjuntos S e T, respetivamente, aos quais foram retirados os elementos da interseção de S e T, tal que S' e T' são não vazios. Como a S e T foram retirados os mesmos elementos, e S e T tinham a mesma soma, S' e T' têm a mesma soma.

Sejam *F(s)* e *F(t)* respetivamente os máximos dos conjuntos $S´$ e $T´$. Como $S´$ e $T´$ não têm elementos em comum, *F(s)* será diferente de *F(t)*. Suponha-se sem perda de generalidade que $F(t) > F(s) \Leftrightarrow F(t) \geq F(s+1)$. $F(s+1)$ será igual à soma de todos os s primeiros termos da sucessão de Fibonacci, e como $S´$ não conterá todos estes termos pois não possui termos consecutivos, então $S´<T´$. Assim, chega-se a uma contradição, pelo que um dos conjuntos será vazio.

Suponha-se sem perda de generalidade que $T´$ é vazio. Então, a soma dos elementos de $T´$ é 0. Logo, a soma de $S´$ é 0, e, como a sucessão de Fibonacci só tem termos positivos, $S´$ será vazio. Assim, $S = T$, logo, apenas existirá uma representação de Zeckendorf para cada número.

Usando o comando *AllTrue*, obteve-se que os primeiros 1000000 de naturais têm representação única, fortalecendo assim a demonstração teórica.

Para a prova experimental desta propriedade, procurou-se provar que não existem dois subconjuntos de elementos não consecutivos cuja soma seja igual. Para tal criou-se a função Soma que recebe uma lista de inteiros e devolve a soma destes.

Além desta, criou-se a função *Listanaoconsequtivos* que recebe uma lista de números inteiros e que devolve *True* se a diferença entre números adjacentes é diferente de 1. Esta função garante que não existem números consecutivos numa lista, desde que esta esteja ordenada.

A função *EscolheListas* recebe um inteiro e devolve os subconjuntos de elementos entre 2 e esse inteiro, garantindo que não existem números consecutivos, uma vez que os subconjuntos estão ordenados. Foi criada a função *Fib* que recebe uma lista de inteiros e devolve os números de Fibonacci de índice correspondente.

Através da amálgama das funções previamente mencionadas, obteve-se a função *SomaListasDeFib* que recebe um inteiro n e devolve a lista das somas de todos os subconjuntos constituídos por números de Fibonacci até ao índice n. Basta então aplicar o comando *DuplicateFeeQ* e observa-se que não existem repetições confirmando, assim, a propriedade.

8 Exercício 8

8.1 Alínea a)

A função *kaprekar* realiza uma iteração do algoritmo de Kaprekar. Assim, ao aplicar um *FixedPoint* do *Mathematica* ao triplo $\{n, 4, 10\}$, para qualquer $n < 10^4$, obtemos a constante 6174.

Note-se que, uma vez que os dígitos são ordenados de forma crescente e decrescente, a ordem com que surgem em n não é relevante para o resultado final. Desta forma, $n = 1$ terá exatamente a mesma iteração que $n = 100$, ou $n = 1000$, por exemplo.

Para verificar que o algoritmo funciona, de facto, para todos os números com 4 dígitos, implementamos um ciclo For, que, dada uma lista l inicialmente vazia, adiciona-lhe o triplo (n,4,10) se $FixedPoint[kaprekar, \{n, 4, 10\}]$ for diferente da constante 6174.

Como observado, o resultado final é uma lista vazia, ou seja, todos os números de 4 dígitos diferentes conduzem à constante 6174.

8.2 Alínea b)

Para esta alínea, criámos a função *constantes*, que, dado um número de dígitos e uma base, calcula as constantes de Kaprekar correspondentes. No entanto, para certos números de dígitos, não existem constantes, mas sim ciclos.

Foi necessário, então, criar uma função *ciclos*, que, dado um triplo $\{n, digitos, base\}$, calcula o ciclo que se obtém das sucessivas iterações do algoritmo de Kaprekar. *Ciclosslist* é uma outra função, implementada com o objetivo de encontrar todos os ciclos associados a um certo número de dígitos.

A função não está totalmente otimizada, uma vez que não reconhece o mesmo ciclo, se este estiver ordenado de forma diferente, o que origina um excesso de resultados, pois repete o mesmo ciclo tantas vezes quanto seu comprimento.