Capítulo 3

Sobre o esquema "Dividir para Conquistar"

3.1 Uma recorrência associada ao esquema "dividir para conquistar"

Suponhamos que uma implementação do esquema genérico "dividir para conquistar" tem a seguinte forma

```
     função f(problema de tamanho n):
       if n=1:
         return <solução imediata>
       else:
(1)      divide-se o problema em b partes de tamanho n/b
         fazem-se a chamadas recursivas
(2)          f(problema de tamanho n/b)
(3)      combinam-se as a soluções
         ... no resultado que é retornado
```

Para simplificar vamos supor que n é potência de b, seja $n = b^m$. Admitindo que a divisão (1) e a combinação (3) são executadas num tempo de ordem $O(n^k)$, temos a seguinte recorrência para a definição de um majorante do tempo de execução

$$t(n) = at(n/b) + cn^k \tag{3.1}$$

Para simplificar supomos que $t(1) = c$. Para estudar a solução desta recorrência, começemos por

considerar o caso $n = b^2$. Temos

$$\begin{aligned} t(n) &= at(n/b) + cn^k && (3.2)\\ &= a(at(n/b^2) + c(n/b)^k) + cn^k && (3.3)\\ &= a^2t(1) + acb^k + cb^{2k} && (3.4)\\ &= a^2c + acb^k + cb^{2k} && (3.5)\end{aligned}$$

Note-se que nesta expressão a, b e k são constantes; a variável é m com $n = b^m$, ou seja $m = \log_b n$.

No caso geral obtemos

$$t(n) = c(a^m + a^{m-1}b^k + \cdots + a^0b^{mk}) = ca^m \sum_{i=0}^{m} \left(\frac{b^k}{a}\right)^i$$

Casos a considerar: $a > b^k$, $a = b^k$, $a < b^k$.

Caso 1. $a > b^k$

Trata-se de uma soma geométrica. Sendo $r = b^k/a$ é $r < 1$ e temos

$$t(n) = ca^m \frac{r^{m+1} - 1}{r - 1} < \frac{ca^m}{1 - b^k/a} \in O(a^m)$$

Mas $a^m = a^{\log_b n} = b^{(\log_b a)(\log_b n)} = n^{\log_b a}$.

Caso 2. $a = b^k$

Neste caso os termos da soma são iguais e $t(n) = c(m+1)a^m \in O(ma^m)$. Como $m = \log_b n$, é

$$n^k = b^{(\log_b n)(log_b a)} = a^{\log_b n}$$

donde

$$ma^m = (\log_b n)a^{\log_b n} = (\log_b n)n^k \in O(n^k \log n)$$

Caso 3. $a < b^k$

Procedendo como no primeiro caso e sendo $r = b^k/a$, temos $r > 1$ e

$$t(n) = ca^m \frac{b^{k(m+1)}/a^{m+1} - 1}{b^k/a - 1} \in O(b^{km}) = O(n^k)$$

Em resumo, temos

Teorema 10 *A solução de uma recorrência com a equação geral da forma $t(n) = at(n/b) + cn^k$ onde c e k são inteiros positivos, a e b são inteiros com $a \geq 1$, $b \geq 2$ tem a seguinte ordem de*

grandeza

$$\begin{cases} t(n) \in O(n^{\log_b a}) & se\ a > b^k \\ t(n) \in O(n^k \log n) & se\ a = b^k \\ t(n) \in O(n^k) & se\ a < b^k \end{cases}$$

O resultado mais forte que resulta de substituir em cima a ordem de grandeza $O(\cdot)$ por $\Theta(\cdot)$ é também válido.

Utilizaremos este resultado diversas vezes neste curso.

Exercício 32 *Considere os algoritmos "merge sort" e "pesquisa binária"; mostre como o resultado anterior pode ser utilizado em cada um desses algoritmos para encontrar a correspondente ordem de grandeza do tempo de execução.*

3.2 Multiplicar mais rapidamente...

Em primeiro lugar vamos considerar a multiplicação de 2 inteiros de n bits, isto é, de grandeza compreendida entre 2^{n-1} e 2^n-1, e depois consideraremos a multiplicação de 2 matrizes quadradas de dimensão $n \times n$. Os resultados serão surpreendentes.

3.2.1 Multiplicação de inteiros

Consideremos 2 inteiros de n bits, ou um pouco mais geralmente, 2 inteiros x e y em que o maior deles tem n bits, $n = \max\{|x|, |x|\}$. Um primeiro método se obter o produto xy consiste em somar y parcelas todas iguais a x,

```
9482 * 2596:     9482   +
                 9482      (2596 parcelas!)
                 ....
                 9482
               ------
             24615272
```

ou, em algoritmo

```
Algoritmo de multiplicação (somas sucessivas)

            prod1(x,y):
              s=0
              for i=1 to y:
                s=s+ x
              return s
```

Este algoritmo é terrivelmente ineficiente, sendo exponencial no tamanho dos dados n: o número de adições é $\Omega(2^n)$ (sendo $|x| = |y| = n$) e a cada adição corresponde um tempo de $\Theta(n)$.

O método "escolar", que todos conhecemos de cor, é muito mais eficiente

```
 9482 * 2596:     9482   x
                  2596
              --------
                 56892
                85338
               47410
              18964
             ---------
              24615272
```

ou, em algoritmo

```
Algoritmo escolar da multiplicação de inteiros.
Representação de b na base 10: b = b[n-1]b[n-2]...b[1]b[0]

            prod2(x,y):
              s=0
              shift=0
              for i=0 to n-1:
                s=s + shift(x*y[i],i) -- shift i posições para a esquerda
              return s
```

(é claro que na base 2, a operação `a*b[i]` é trivial) Este algoritmo é muito mais eficiente, na realidade de ordem $\Theta(n^2)$ uma vez que para cada `i` são efectuadas n adições (cada uma em tempo $\Theta(n)$) mais um "shift" que não altera a ordem de grandeza.

Será este o método mais eficiente? Vamos mostrar que não. Em 1962 o russo Anatoli Karatsuba descobriu um método mais eficiente (com uma ordem de grandeza inferior a n^2), baseado na ideia de "dividir para conquistar". Consideremos os inteiros x e y com (não mais de) $2n$ bits

$$\begin{cases} x = a \times 2^n + b \\ y = c \times 2^n + d \end{cases}$$

Os primeiros n bits de x "estão em" a e os últimos n em b; e analogamente para y. Efectuemos o produto xy:

$$xy = ac2^{2n} + bc2^n + ad2^n + bd \tag{3.6}$$

Na realidade, esta igualdade não nos vai ajudar muito em termos de eficiência: para multiplicar 2 números de $2n$ bits temos de efectuar 4 multiplicações de inteiros de n bits, isto é, vamos ter um tempo de execução definido pela recorrência

$$\begin{cases} t(1) = k \\ t(2n) = 4t(n) + cn \end{cases}$$

onde k e c são constantes. O termo cn representa um majorante do tempo de execução das 3 adições, dos "shifts" bem como do tempo de procesamento dos "carries". A expressão anterior corresponderá a uma multiplicação do tipo

```
          a   b
          c   d
     -----------
         ac   bd
     bd  bc
  --------------
    ...  ...  ...
```

Exercício 33 *Mostre que a solução da recorrência anterior é de ordem* $\Theta(n^2)$.

Então, relativamente à ordem de grandeza do algoritmo "escolar", nada se ganhou com esta aplicação do "dividir para conquistar"! A melhoria da eficiência assimptótica resulta da observação crucial seguinte: é possível reescrever a igualdade (3.6) usando apenas 3 multiplicações de inteiros de n (ou $n+1$) bits

$$xy = ac(2^{2n} - 2^n) + (a+b)(c+d)2^n + bd(1 - 2^n) \tag{3.7}$$

Pode-se, efectuando as multiplicações de (3.7), mostrar a equivalência a (3.7). E agora temos apenas 3 multiplicações, para além de diversas operações ("shifts" e adições) implementáveis em tempo linear. Temos então um tempo de execução definido pela recorrência seguinte (onde c é uma nova constante):

$$\begin{cases} t(1) = k \\ t(2n) = 3t(n) + cn \end{cases}$$

Exercício 34 *Mostre que a solução da recorrência anterior é de ordem* $\Theta(n^{\log_2 3}) = \Theta(n^{1.5849\ldots})$.

Este método é assintóticamente mais rápido, embora o "overhead" – representado pelas constantes que as ordens de grandeza escondem – seja maior. Só vale a pena aplicar este método quando os inteiros envolvidos são muito grandes.

Será este o método mais rápido que existe para a multiplicação de inteiros? Não. O algoritmo FFT ("Fast Fourier Transform") foi usado por Karp para mostrar que é possível multiplicar 2 inteiros de n bits em tempo $\Theta(n \log^2(n))$ e mais tarde (em 1971) Schönhage e Strassen descobriram um algoritmo de eficiência $\Theta(n \log n \log(\log n))$ que é, em termos de ordem de grandeza, o algoritmo de multiplicar mais rápido que se conhece.

3.2.2 Multiplicação de matrizes

A história aqui vai ser semelhante à da secção anterior. Consideramos a multiplicação de 2 matrizes quadradas de dimensão $n \times n$ e vamos utilizar o modelo uniforme, supondo que cada adição ou multiplicação elementar é efectuada em tempo $O(1)$; trata-se de uma hipótese realista se os inteiros contidos nas matrizes não são muito grandes (cabem numa "palavras" do computador).

Tal como na secção anterior vamos ver que uma aplicação elementar do esquema "dividir para conquistar" não leva a qualquer redução da ordem de grandeza, sendo necessário usar observações não triviais para haver sucesso. Esses "feitos" são: na secção anterior (multiplicação de inteiros grandes) conseguir escrever uma expressão com apenas 3 (em vez de 4) multiplicações de inteiros e na presente secção a redução de 8 para 7 do número de multiplicações matriciais numa determinada expressão.

Exercício 35 *Mostre que no algoritmo usual de multiplicação de 2 matrizes quadradas de dimensão* $n \times n$

- *O número de multiplicações elementares é* n^3
- *O número de adições elementares é* $n^2(n-1)$
- *Conclua que, usando o modelo uniforme, a complexidade do algoritmo usual é* $\Theta(n^3)$.

Se dividirmos cada uma das matrizes de dimensão $2n \times 2n$ em 4 sub-matrizes, temos que efectuar o produto

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix} \times \begin{bmatrix} E & F \\ G & H \end{bmatrix} = \begin{bmatrix} AE+BG & AF+BH \\ CE+DG & CF+DH \end{bmatrix} \tag{3.8}$$

As operações que levam à matriz resultado

$$R = \begin{bmatrix} R_{11} & R_{12} \\ R_{21} & R_{22} \end{bmatrix}$$

são pois as seguintes

$$\begin{cases} R_{11} &=& AE+BG \\ R_{12} &=& AF+BH \\ R_{21} &=& CE+DG \\ R_{22} &=& CF+DH \end{cases} \tag{3.9}$$

As adições são efectuadas em tempo $\Theta(n^2)$, havendo 8 chamadas recursivas; assim o tempo de execução é definido por

$$\begin{cases} t(1) = k \\ t(2n) = 8t(n) + cn^2 \end{cases}$$

Exercício 36 *Mostre que a solução $t(n)$ desta recorrência satisfaz $t(n) \in \Theta(n^3)$.*

Não se ganhou nada em termos de tempo de execução. Nem sequer na realidade se mudou basicamente o algoritmo.

Notemos que num algoritmo recursivo implementado segundo um esquema baseado em (3.8) apenas vai haver recursão na multiplicação de matrizes, sendo as adições efectuadas imediatamente; a ordem de grandeza do tempo das adições é $\Theta(n^2)$. Strassen mostrou que o produto 3.8 pode ser efectuado com apenas 7 multiplicações (em vez de 8). As operações que levam à matriz resultado R são as seguintes (confirme!) onde os M_i são matrizes temporárias

$$\begin{cases} M_1 &= (A+D)(E+H) \\ M_2 &= D(G-E) \\ M_3 &= (B-D)(G+H) \\ M_4 &= (A+B)H \\ M_5 &= (C+D)E \\ M_6 &= A(F-H) \\ M_7 &= (C-A)(E+F) \\ R_{11} &= M_1+M_2+M_3-M_4 \\ R_{12} &= M_4+M_6 \\ R_{21} &= M_2+M_5 \\ R_{22} &= M_1-M_5+M_6+M_7 \end{cases}$$

Compare-se com (3.9) e repare-se que agora há apenas 7 multiplicações de matrizes de $n \times n$. O tempo de execução é pois da ordem de grandeza definida pela seguinte recorrência

$$\begin{cases} t(1) = 1 \\ t(2n) = 7t(n) + cn^2 \end{cases}$$

Exercício 37 *Mostre que a solução desta recorrência é de ordem* $\Theta(n^{\log 7})$.

O algoritmo de Strassen só é vantajoso na prática para o cálculo do produto de matrizes de grande dimensão.

O algoritmo de Strassen é o mais rápido? Mais uma vez, não. Tem havido diversas melhorias em termos de ordens de grandeza, sendo actualmente o algoritmo de Coppersmith e Winograd o mais rápido.

Há ainda um aspecto que é importante mencionar. Nas implementações concretas, quando as chamadas recursivas chegam a uma dimensão n suficientemente pequena recorre-se ao algoritmo clássico de ordem n^3, pois para pequenos valores de n este é muito mais rápido. Por outro lado, os algoritmos pós-Strassen têm “constantes escondidas” tão grandes que se tornam impraticáveis; os valores de n para os quais seriam mais rápidos são gigantescos. Em conclusão: as ordens de grandeza não nos “contam a história toda”. Por exemplo, se os tempos de execução dos algoritmos A e B são respectivamente n^2 e $150n^{2.8}$, embora B seja assintóticamente mais rápido que A, isso só acontece para valores de n superiores a $150^5 \approx 76\,000\,000\,000$!

Capítulo 4

Algoritmos aleatorizados e classes de complexidade

Neste capítulo vamos ver que, mesmo para problemas que não têm nada a ver com probabilidades ou estatística, como por exemplo o problema da ordenação de um vector, o acesso a números aleatórios pode ser benéfico, no sentido em que se obtêm algoritmos mais eficientes (no caso médio), independentemente da distribuição dos dados. Ilustraremos as vantagens da aleatorização dos algoritmos com 2 exemplos: um algoritmo de coloração (página 57), um algoritmo que testa uma dada matriz é o produto de outras 2 matrizes (página 59) e ainda o algoritmo "quick sort" (página 60). Em seguida (página 73), classificaremos os algoritmos aleatorizados e definiremos classes de complexidade baseadas em máquinas de Turing que têm acesso a uma sequência de bits aleatórios. O modo de obter inteiros verdadeiramente aleatórios ou então "pseudo-aleatórios" e o modo como os segundos podem aproximar ou não os primeiros, são questões (interessantes!) que não são tratadas nesta secção.

Em secções posteriores apresentamos um teste aleatorizado de primalidade devido a Rabin e Miller e discutiremos em termos mais gerais os algoritmos aleatorizados e as classes de complexidade aleatorizadas.

4.1 Um problema de coloração

Vamos agora considerar o seguinte problema:

Problema "colorir". É dado um conjunto S com n elementos e uma família A_1, A_2,..., A_m de sub-conjuntos distintos de S, cada uma contendo exactamente r elementos. Pretende-se atribuir a cada elemento de S uma de 2 cores (azul e vermelho, por exemplo) por forma a que em cada sub-conjunto A_i exista pelo menos um elemento com uma das cores e um elemento com a outra cor.

Note-se que cada elemento de S pode pertencer a mais que uma das famílias A_i (pode até pertencer a todas). Por exemplo, S poderia ser um conjunto de alunos, já inscritos em disciplinas A_i (de forma que cada turma tenha exactamente r alunos inscritos) e pretende-se entregar a cada aluno um computador ou uma impressora (mas não as 2 coisas!) por forma que em cada disciplina exista pelo menos um aluno com um computador e pelo menos um aluno com uma impressora.

Nota importante. Para que o algoritmo aleatorizado que vamos definir funcione convenientemente, vamos supor que $m \leq 2^{r-2}$.

A primeira ideia para resolver este problema leva geralmente a um algoritmo bastante complicado. Mas acontece que o seguinte algoritmo aleatorizado resolve o problema em pelo menos 50% dos casos:

```
Algoritmo aleatorizado básico para o problema da coloração
----------------------------------------------------------
Para cada elemento de x de S:
  atribuia-se a x, aleatoriamente e uniformemente (prob=1/2),
    a cor azul ou a cor vermelha
Verifique-se se a coloração está correcta
Se estiver: retorna a solução, senão retorna "não definido"
```

Nada mais simples! É claro que este algoritmo pode errar (ou melhor, retornar `"não definido"`), mas vamos ver que a probabilidade de isso acontecer não ultrapassa 1/2. Vamos demonstrar esse facto.

Mas vejamos primeiro um exemplo: seja $n = 9$, $m = 4$ e $r = 4$ (verifica-se pois a condição $m \leq 2^{r-2}$) e os conjuntos

$$A_1 = \{1, 2, 3, 4\},\ A_2 = \{3, 5, 8, 9\},\ A_3 = \{1, 5, 6, 7\},\ A_4 = \{2, 3, 6, 9\}$$

Atirei a moeda ao ar e obtive a coloração: 1, 2, 3, 4, 5, 6, 7, 8, 9 (para quem está a ler a preto e branco: 2, 3, 4, 6, 7 e 8 são vermelhos e os restantes azuis), É fácil verificar (verifique!) que tivemos sorte, pois obtivemos uma solução.

Para qualquer i com $1 \leq i \leq m$ a probabilidade de todos os elementos do conjunto A_i terem a mesma cor é 2^{-r+1} (porquê?). E a probabilidade de pelo menos um A_i não ser válido, isto é,

de todos os seus elementos terem a mesma cor, não excede $m2^{-r+1}$ (porquê?). Assim, e usando a hipótese $m \leq 2^{r-2}$, concluimos que a probabilidade da coloração "à sorte" não estar correcta não excede

$$m2^{-r+1} \leq 2^{r-2}2^{-r+1} = \frac{1}{2}$$

E o que é que obtivemos? *Um algoritmo aleatorizado que, independentemente dos dados, acerta em pelo menos 50% das vezes.*

Claro que 50% é pouco, mas vamos ver na Secção 4.4 como tornear o problema.

4.2 O produto de duas matrizes iguala uma terceira?

Suponhamos que conhecemos 3 matrizes quadradas A, B e C, todas de dimensões $n \times n$, e que pretendemos saber se $AB = C$. O método mais imediato consiste em multiplicar A por B e ver se o resultado é igual a C. Se usarmos o algoritmo clássico de multiplicação de matrizes, a complexidade deste método é de ordem $\Theta(n^3)$. Mesmo utilizando os métodos assimptoticamente mais rápidos, não conseguimos melhor que $\Theta(n^{2.373\cdots})$.

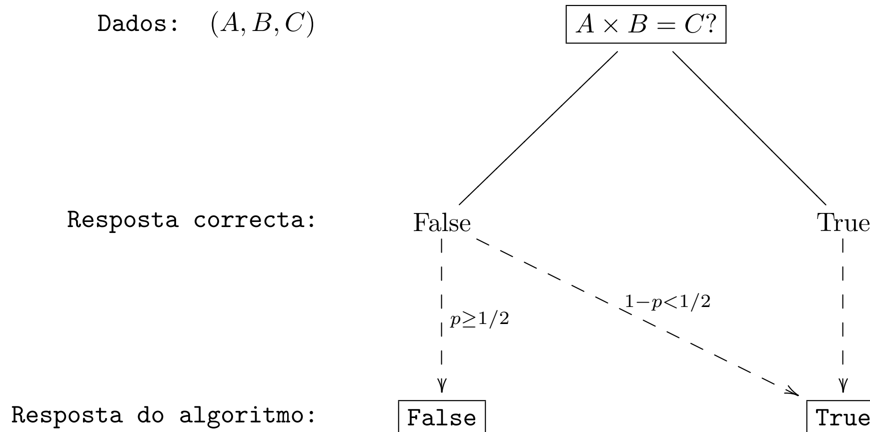
Vamos apresentar um algoritmo aleatorizado (que pode errar) que é muito simples e rápido.

```
Algoritmo que testa se A*B=C
----------------------------

  Dados: Matrizes quadradas, n*n, A, B, C
  Resultado: True se A*B=C, False caso contrário

1    - Gera-se um vector coluna r com n elementos em que cada elemento
       é 0 ou 1 com probabilidade 1/2 (de forma independente)
2    - Calcula-se x=Br (vector coluna) e depois y=Ax (vector coluna)
3    - Calcula-se z=Cr (vector coluna)
4    - Se y=z: retorna True
       senão:  retorna False
```

A ordem de grandeza do tempo de execução é $\Theta(n^2)$, sendo dominada pelas produtos matriciais (linhas 2 e 3). Independentemente dos dados, temos

$$\begin{cases} \text{se } AB = C & \rightarrow \quad \text{resposta } \texttt{True} \\ \text{se } AB \neq C & \rightarrow \quad \text{resposta } \texttt{False} \text{ com probabilidade } \geq 1/2 \end{cases}$$

No diagrama seguinte representamos por p a probabilidade da resposta do algoritmo ser `False`.



Notas.

– Este algoritmo funciona em qualquer corpo $(A, +, \times)$.

– Embora a probabilidade de o algoritmo errar (isto é, responder `True` quando $A \times B \neq C$) possa ser relativamente grande, a repetição da execução do algoritmo para os mesmos dados permite reduzir esse erro a valores exponencialmente pequenos; mais especificamente, com k execuções do algoritmo, a probabilidade de erro não excede 2^{-k}. Ver a Secção 4.4.

4.3 O "quick sort"

Começaremos por analisar o caso médio e o pior caso do algoritmo clássico do "quick sort" e depois uma versão aleatorizada desse algoritmo em que o pivot é escolhido (de cada vez que o algoritmo corre) de forma aleatória.

4.3.1 Esquema básico do "quick sort"

Vamos supor que todos os elementos do vector a ordenar são distintos; se tal não acontecer, é fácil alterar o algoritmo e a sua análise por forma a poder haver elementos iguais.

```
Quick sort clássico
-------------------
Dados: um vector x[a..b] a ordenar

1     Se o número de elementos do vector (b-a+1) é 0 ou 1:
        HALT
2     Escolhe um "pivot", isto é um dos x[p], com a <= p <= b
      Seja piv=x[p]
3     (Split) particiona-se o vector x[a..b] em 3 partes:
        x[a..m-1]: com elementos x[i]<piv
        x[m]:      com o valor piv, x[m]=piv
        x[m+1..b]: com elementos x[i]>piv
4     Recursivamente aplica-se o algoritmo a x[a..m-1]
5     Recursivamente aplica-se o algoritmo a x[m+1..b]
```

Não se trata ainda de um um algoritmo[1], uma vez que a parte de particionamento do vector (linha 3) não está especificada. Mas é suficiente para os nossos propósitos.

4.3.2 Análise no pior caso do "quick sort" clássico.

Vamos supor que escolhemos como pivot o primeiro elemento, `piv=x[1]`. Se fosse escolhido qualquer outro elemento de índice fixo como pivot, as conclusões seriam semelhantes.

Seja `n=b-a+1` o número de elemento do vector. Se o vector já estiver ordenado, `x[a]`<`x[a+1]`<...<`x[b]`, após a primeira partição fica o lado esquerdo do vector (elementos menores que `piv`) vazio e o lado direiro com `n-1` elementos. A partição demora tempo $\Theta(n)$ (efectuam-se $n-1$ comparações). Em seguida efectuam-se chamadas recursivas a vectores de tamanhos `n-1`, `n-2`,..., 2. O tempo total (e o número de comparações é pois de ordem $\Theta(n^2)$. O número exacto de comparações é $(n-1)+(n-2)+\ldots+1 = n(n-1)/2$. Obviamente há outros casos com este mau comportamento (em termos da ordem de grandeza), como o do vector `[1,8,2,7,3,6,4,5]`; o que não há é casos com ordem de grandeza pior que $\Theta(n^2)$, uma vez que a linha 2 (escolha do pivot) pode ser efectuada no máximo `n-1` vezes (de cada vez que é efectuada há necessariamente menos um elemento, o pivot, que não faz parte dos sub-vectores a ordenar) e de cada vez na linha 3 fazem-se no máximo `n-1` comparações.

Em conclusão, no pior caso, o "quick sort" é de ordem $\Theta(n^2)$.

4.3.3 Análise do tempo médio do "quick sort" clássico

Assumimos aqui que o aluno tem conhecimentos básicos de probabilidades. Por simplicidade, continuamos a supor que todos os elementos do vector são distintos.

[1] Em certas linguagens de "alto nível" é possível escrever o "quick sort" de forma curta e elegante. Tais linguagens não existiam ainda quando este algoritmo foi inventado por C. A. R. Hoare em 1960! Num dos apêndices (pág. 166) pode ver-se implementações do "quick sort" em Haskell, Prolog e Python.

Ao particionar o vector com `n` elementos fazem-se `n - 1` comparações, uma vez que o pivot é comparado com cada um dos outros elementos para definir o sub-vector da esquerda e o sub-vector da direita. Dependendo do valor do pivot, podem acontecer os seguintes casos

Num. de elementos à esquerda	Num. de elementos à direita
0	$n-1$
1	$n-2$
2	$n-3$
...	...
$n-2$	1
$n-1$	0

Como o pivot (o primeiro elemento do vector) tem igual probabilidade de ser o menor, o segundo menor,..., o maior, qualquer uma das partições da tabela anterior tem a mesma probabilidade, $1/n$. Então a recorrência que define o *valor esperado* do número de comparações é a seguinte

$$\begin{cases} t(0) = 0 \\ t(1) = 0 \\ t(n) = (n-1) + \frac{1}{n}\sum_{i=0}^{n-1}(t(i) + t(n-i-1)) \end{cases}$$

Note-se que $t(0) = 0$ e que cada um dos restantes $t(i)$ ocorre duplicado. Por exemplo, para $n = 8$, aparece o somatório

$$\begin{aligned} t(4) &= 3 + [(t(0)+t(4)) + (t(1)+t(3)) + (t(2)+t(2)) + (t(3)+t(1)) + (t(4)+t(0))] \\ &= 3 + 2[(t(1)+t(2)+t(3)+t(4))] \end{aligned}$$

Assim a equação geral da recorrência pode escrever-se

$$t(n) = (n-1) + \frac{2}{n}\sum_{i=1}^{n-1} t(i)$$

A solução desta recorrência vai seguir os 2 passos tradicionais: inspiração e demonstração.

– *Inspiração.* Vamos admitir que em quase todos os casos o particionamento é tal que a parte esquerda e a parte direita são sensivelmente iguais. Se fossem sempre exactamente iguais[2], a altura máxima da árvore associada às chamadas recursivas não inferior a $\log n$. Por outro lado, *em cada nível de recursão* o número total de comparações não pode exceder n (aliás é sempre inferior a n). Assim, esperamos que, eventualmente, o número total de comparações

[2] A altura mínima ocorre quando n for da forma $n = 2^p - 1$ e os pivots são sempre a mediana do sub-vectores a ordenar.

seja limitado por $cn \log n$ para alguma constante $c > 0$. A seguir vamos demonstrá-lo.

– *Demonstração.*

Pretendemos mostrar que o valor $t(n)$ definido pela recorrência satisfaz $t(n) \leq cn \ln n$ para uma constante c a determinar; é mais conveniente usar o logaritmo natural e, dado o facto de termos mais tarde que determinar a constante c por forma a que a proposição seja verdadeira, a base do logaritmo é indiferente. Os casos base são triviais. Para o passo de indução temos

$$\begin{aligned}
t(n) &= (n-1) + \frac{2}{n}\sum_{i=1}^{n-1} t(i) && (4.1)\\
&\leq (n-1) + \frac{2c}{n}\sum_{i=1}^{n-1} i \ln i && (4.2)\\
&\leq (n-1) + \frac{2c}{n}\int_1^n x \ln x dx && (4.3)\\
&\leq (n-1) + \frac{2c}{n}\left(\frac{1}{2}n^2 \ln n - \frac{n^2}{4} + \frac{1}{4}\right) && (4.4)\\
&\leq cn \ln n && (4.5)
\end{aligned}$$

desde que $c \geq 2$. Os passos que foram usados são

– Da linha (4.1) para a linha (4.2): hipótese indutiva.

– Da linha (4.2) para a linha (4.3): se $f(x)$ é positivo e crescente em $[1, n]$ então a soma $f(1)+f(2)+\ldots+f(n-1)$ é majorada pelo integral $\int_1^n f(x)$ (basta usar a interpretação dos integrais definidos como áreas).

– Da linha (4.3) para a linha (4.4): cálculo do integral definido.

– Da linha (4.4) para a linha (4.5): propriedade fácil de verificar.

Assim, podemos enunciar o seguinte resultado

Teorema 11 *Supondo-se que todas as permutações do vector inicial são igualmente prováveis, o valor esperado do número de comparações efectuadas no algoritmo clássico do "quick sort" não excede* $2n \ln n \approx 1.386 n \log n$*, tendo pois o algoritmo um tempo médio de execução deordem* $O(n \log n)$.

4.3.4 O "quick sort" aleatorizado

Consideremos agora uma pequena alteração ao "quick sort" clássico: o índice correspondente ao elemento pivot é escolhido aleatóriamente de cada vez que o algoritmo corre e de forma (estatis-

ticamente) independente da distribuição dos dados[3]

```
Quick sort aleatorizado
-----------------------
Dados: um vector x[a..b] a ordenar

1    Se o número de elementos do vector (b-a+1) é 0 ou 1:
       HALT
2    É escolhido aleatoriamente e uniformemente (com igual
     probabilidade) um índice p de {a,a+1,...,b}.
     Seja piv=x[p]
3    Particiona-se o vector x[a..b] em 3 partes:
       x[a..m-1]: com elementos x[i]<piv
       x[m]:      com o valor piv, x[m]=piv
       x[m+1..b]: com elementos x[i]>piv
4    Recursivamente aplica-se o algoritmo a x[a..m-1]
5    Recursivamente aplica-se o algoritmo a x[m+1..b]
```

A alteração é mínima, mas os efeitos na complexidade são grandes.

Consideremos um qualquer vector dado `v[]` fixo. Toda a análise que se fez na secção anterior (páginas 61 a 63) continua válida uma vez que, devido à linha 2 do algoritmo, qualquer índice tem a mesma probabilidade de ser escolhido como índice do pivot. Temos assim o resultado seguinte:

Teorema 12 *Para qualquer vector dado fixo, o valor esperado do número de comparações efectuadas pelo algoritmo "quick sort" aleatorizado não excede* $2n \ln n \approx 1.386 n \log n$*, tendo pois o algoritmo um tempo médio de execução de ordem* $O(n \log n)$.

Obviamente, e como corolário deste teorema, se o vector dado não for fixo mas obedecer a uma distribuição probabilística qualquer, continua a verificar-se o resultado anterior.

Corolário 1 *Para qualquer distribuição probabilística do vector dado, o valor esperado do número de comparações efectuadas pelo algoritmo "quick sort" aleatorizado não excede* $2n \ln n \approx 1.386 n \log n$*, tendo pois o algoritmo um tempo médio de execução deordem* $O(n \log n)$.

O algoritmo clássico e o algoritmo aleatorizado são quase iguais; no último o modelo de computação (o computador) é probabilístico pois tem acesso a uma fonte de números aleatórios. Em termos de variáveis aleatórias, a variável aleatória Y associada ao resultado y é função de 2 variáveis aleatórias independentes:

– A variável aleatória X associada ao vector dado x

– A variável aleatória R associada ao modelo de computação que deve ser necessariamente independente de Y e uniforme.

[3]Vamos continuar a supor que todos os elementos do vector a ordenar são distintos.

E o que é que obtivemos com a aleatorização do algoritmo?

> Qualquer que seja a distribuição probabilística associada aos dados (incluindo mesmo casos extremos como o caso em que a probabilidade é 0 para todos os vectores possíveis excepto para um) o tempo médio de execução do algoritmo é de ordem $O(n \log n)$.
> Por outras palavras, em termos do tempo médio, não existem más distribuições dos dados.

Comparemos com o que se passa com o algoritmo clássico:

> Embora o tempo médio de execução seja de ordem $\Theta(n \log n)$, qualquer que seja o pivot escolhido (fixo) existem vectores (e portanto existem distribuições probabilísticas dos dados) para os quais o tempo médio de execução do algoritmo é de ordem $O(n^2)$.

4.4 Técnica de redução da probabilidade de erro

Vamos apresentar uma técnica de reduzir a probabilidade de erro de um algoritmo aleatorizado; essa técnica consiste essencialmente na repetição do algoritmo. Embora este método seja bastante geral, será explicado com o problema da secção anterior (problema de coloração).

Suponhamos que corremos o algoritmo 2 vezes. Temos 2 colorações. Qual a probabilidade de pelo menos uma estar correcta? A probabilidade de estarem as 2 erradas é não superior a[4]

$$\frac{1}{2} \times \frac{1}{2} = \frac{1}{4}$$

Assim, a probabilidade de pelo menos uma estar correcta é pelo menos $1 - 1/4 = 3/4$. Mais geralmente, se corrermos o algoritmo k vezes, a probabilidade de pelo menos numa das vezes o algoritmo produzir uma solução (correcta) é não inferior a $1 - 1/2^k$.

Podemos então escolher uma das 2 hipóteses seguintes

– Correr o algoritmo, digamos um máximo de 500 vezes. A probabilidade de não ser obtida uma solução correcta é não superior a $2^{-500} \approx 3 \times 10^{-151}$, um valor ridiculamente pequeno, igual a 0 para todos os efeitos práticos[5]. No algoritmo seguinte, o algoritmo aleatorizado básico de coloração (página 58) é executado no máximo k vezes, onde k é um parâmetro fixo. A probabilidade de ser retornada uma solução é pelo menos $1 - 2^{-k}$.

[4] Uma vez que a probabilidade da conjunção de 2 acontecimentos independentes é o produto das respectivas probabilidades.
[5] "Quem acha que 10^{100} não é infinito deve ter perdido o seu juizo", John von Neumann.

```
Algoritmo aleatorizado para o problema da coloração
com probabilidade exponencialmente pequena de não ser retornada uma
solução.
--------------------------------------------------------------------
Seja COL o algoritmo aleatorizado básico para o problema da coloração
--------------------------------------------------------------------

  for i=1 to n:
      Execute COL(S,A1,A2,...Am)
      se é retornada uma solução s:
         return s
  return "não definido"
```

– Correr o algoritmo até ser produzida uma solução. O número de vezes que é necessário correr o algoritmo para ser obtida uma solução é uma variável aleatória com um valor médio muito pequeno

$$\frac{1}{2} + 2\frac{1}{4} + 3\frac{1}{8} + \ldots = 2$$

4.5 Outro algoritmo aleatorizado: o algoritmo de Rabin-Miller

O problema de determinar se um inteiro n é primo, o chamado problema da primalidade, é um problema básico na Teoria dos Números e tem aplicações muito importantes na área da criptografia. Assim por exemplo, os 2 factos seguintes

– A existência de algoritmos eficientes para o problema da primalidade

– A densidade relativamente elevada dos primos[6]

permitem a geração de números primos grandes com razoável eficiência, tal como se exige no protocolo RSA.

4.5.1 Ineficiência dos algoritmos elementares de primalidade

Os algoritmos elementares de primalidade são exponenciais, ver o exercício seguinte.

Exercício 38 *Mostre que os seguintes algoritmos são exponenciais (em $|n| \approx \log n$)*

a) Testar se cada um dos inteiros $2, 3, \ldots, \lfloor n/2 \rfloor$ *é divisor de* n.

[6] Teorema. . .

a) Testar se cada um dos inteiros $2, 3, \ldots, \lfloor\sqrt{n}\rfloor$ *é divisor de* n.

Nota. *Mesmo se nos limitarmos a testar como possíveis divisores os primos não superiores a* $\lfloor\sqrt{n}\rfloor$*, o algoritmo continua a não ser polinomial.*

4.5.2 Existem algoritmos polinomiais para a primalidade

Apenas recentemente[7] se demostrou que o problema da primalidade pertence a classe **P**, isto é, que pode ser resolvido por um algoritmo determinístico em tempo polinomial, no pior caso. Contudo, o algoritmo proposto neste trabalho não se usa na prática pois, embora seja polinomial, envolve constantes multiplicativas muito grandes e é de ordem elevada.

4.5.3 Testemunhos rarefeitos da não primalidade

Um número que não é primo diz-se *composto*. Os números compostos têm testemunhos. Designamos por *testemunho* de um determinado facto uma informação (por exemplo, uma palavra ou um inteiro) que permite verificar a validade desse facto em tempo polinomial. Já encontrámos esta noção quando estudámos a classe de complexidade **NP**. Um possível testemunho de um inteiro n ser composto é um divisor não trivial de n. Baseado nesses testemunhos, podemos definir o seguinte algoritmo aleatorizado para determinar se um número é primo

```
 Algoritmo aleatorizado para o problema da primalidade
 Dado: n
 Resultado:
    se n é primo: resposta SIM
    se n é composto: a resposta pode ser NÃO (correcta)
                     mas também pode ser SIM (errada)

(*) m = inteiro aleatório uniforme em {2,3,...,n-1}
   se m divide n:
      return NÂO
   senão:
      return SIM
```

O problema deste algoritmo é que a probabilidade de erro pode ser exponencialmente próxima de 1, não podendo ser limitada superiormente por uma constante inferior a 1; caso contrário, seria possível, por repetição das experiências (correr o algoritmo várias vezes com o mesmo n) reduzir a probabilidade de erro a um valor exponencialmente baixo; quando dizemos “exponencialmente”, estamos a falar em função do número de vezes que o algoritmo corre, ver a Secção 4.4. Porque é

[7] Manindra Agrawal, Neeraj Kayal and Nitin Saxena, *PRIMES is in P* foi publicado um algoritmo polinomial (determinístico) para o teste da primalidade.

que não é possível limitar superiomente o erro por uma constante menor que 1? Consideremos por exemplo, a infinidade de inteiros da forma p^2 em que p é primo. De entre o número exponencial de inteiros que podem ser gerados na linha `(*)` do algoritmo anterior, apenas um, nomeadamente p, é testemunha de o número ser composto; assim, em determinados casos a probabilidade de erro é demasiado grande (da forma $1 - \Theta(2^{-n})$).

O que necessitamos é de outros testemunhos da não primalidade que sejam sempre frequentes, isto é, tais que (dando um possível exemplo) qualquer que seja n não primo, mais de metade dos inteiros compreendidos entre 1 e $n - 1$ sejam testemunhos. Se conseguirmos esta densidade de testemunhos, o seguinte algoritmo

```
 Forma do algoritmo aleatorizado que se pretende
 Dado: n
 Resultado:
    se n é primo: resposta SIM
    se n é composto: resposta incorrecta com probabilidade <= 1/2

(*) m = inteiro aleatório uniforme (possível testemunha) ...
   se m é testemunha:
      return NÃO
   senão:
      return SIM
```

4.5.4 Testemunhos frequentes da não primalidade

Para definirmos testemunhos da não primalidade que sejam frequentes, qualquer que seja o inteiro composto, vamo-nos socorrer do chamado "pequeno teorema de Fermat".

Teorema 13 *Seja p um inteiro primo. Então, qualquer que seja o inteiro a com $1 \leq a \leq p - 1$ é $a^{p-1} = 1 \pmod p$.*

Vejamos o exemplo $p = 5$; temos $1^4 = 1$, $2^4 = 16$, $3^4 = 81$, $4^4 = 256$. O resto da divisão por 5 de todos estes inteiros é 1.

Dem. Consideremos os sucessivos produtos (em $\mathbb{Z}_p$)

$$a,\ 2a,\ \ldots,\ (p-1)a$$

Se 2 destes produtos fossem iguais, seja $\alpha a = \beta a \pmod p$ e teríamos $\alpha = \beta \pmod p$, o que é falso uma vez que o maior divisor comum entre p e qualquer dos coeficientes 1, 2, $p - 1$ é 1. Assim, aqueles $p - 1$ produtos são congruentes, numa qualquer ordem, com[8] $1,\ 2, \ldots,\ p - 1$. Então,

[8] No exemplo que antecede esta prova (em que $a = 4$ e $p = 5$), os sucessivos valores de ka módulo p são sucessivamente 4, 3, 2 e 1.

multiplicando as $p-1$ congruencias temos

$$\begin{aligned} a \times 2a \times \ldots \times (p-1)a &= 1 \times 2 \times \ldots \times (p-1) \pmod p \\ a^{p-1}(p-1)! &= (p-1)! \pmod p \\ a^{p-1} &= 1 \pmod p \end{aligned}$$

A passagem da primeira equação para a segunda resulta da observação feita acima, enquanto a terceira equação se obtém da segunda, dividindo (em $\mathbb{Z}_p$) por $(p-1)!$; note-se (verifique), que $p-1$ não é nulo. □

Assim, temos um novo algoritmo aleatorizado, baseado no novo testemunho de não-primalidade de n: um inteiro a, compreendido entre 2 e $n-1$ tal que $a^{n-1} \neq 1 \pmod p$.

```
 Algoritmo aleatorizado para o problema da primalidade
 Dado: n
 Resultado:
    se n é primo: resposta SIM
    se n é composto: resposta incorrecta com probabilidade ???
                     (a determinar)

(*)  m = inteiro aleatório uniforme em {2,3,...,n-1}
(+)  x = m^(n-1)  (mod n)

     se x=1:
        return PRIMO
     senão:
        return COMPOSTO
```

Para este algoritmo ser eficiente, há que efectuar a potência (linha `(+)`) de forma eficiente. Isso exige

1. Sempre que, no cálculo da potência, se efectua um produto deve-se tomar o resto da divisão por n. Isso evita que haja inteiros de comprimento descomunal e torna razoável o modelo uniforme (uma vez tomando o comprimento máximo da representação de n como parâmetro).
2. O cálculo da potência deve ser feito de modo eficiente. Não basta, por exemplo, efectuar $n-1$ multiplicações por m, o que seria exponencial. Um algoritmo apropriado para a potenciação (escrito em C, para variar) pode ver-se no Apêndice da página 167.

Agora põe-se o problema: será que, sempre que um inteiro n é composto, a frequência destes testemunhos, isto é o quociente

$$\frac{\#\{m : m^{n-1} \pmod n \neq 1\}}{n-2}$$

é, por exemplo, maior ou igual a 0.5 (ou outra constante positiva)?

Fazendo um teste computacional, procurámos inteiros compostosnão superiores a 1000 tais que a frequência dos testemunhos é inferior a 0.5. Quantos encontrámos? Um, 561. Encontramos realmente testemunhos frequentes dos números compostos que funcionam,... quase sempre. O problema é o "quase". Os primeiros inteiros compostos com frequência inferior a 0.5 que encontrámos são

Inteiro	Frequência
561	0.428
1105	0.304
1729	0.250
2465	0.273
2821	0.234
6601	0, 200

Estes inteiros têm um nome, os números de Carmichael, ou pseudo-primos de Fermat. Eles arruinaram o nosso algoritmo aleatórizado de primalidade! Mais informação sobre estes inteiros pode encontrar-se em diversos livros de Teoria dos Números e, por exemplo, em
`http://en.wikipedia.org/wiki/Carmichael_number`.

Felizmente, existe um outro teste que completa este, por forma a que a frequência dos testemunhos seja pelo menos 0.5 (na realidade, pelo menos 0.75).

Teorema 14 *Seja p um primo ímpar, $p - 1 = 2^t u$ com u ímpar e seja $1 \leq a \leq p - 1$. Então verifica-se uma das proposições seguintes*

a) $a^u = 1 \pmod p$

b) Existe um inteiro i com $1 \leq i \leq t - 1$ tal que $a^{2^i u} = -1 \pmod p$.

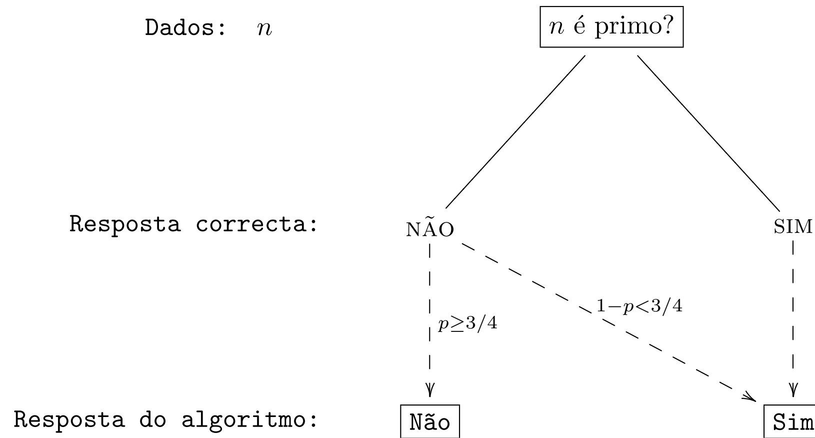
Teorema 15 *Seja n um inteiro composto ímpar, $n - 1 = 2^t u$ com u ímpar.*
Se um inteiro $a \in \{2, 3, \ldots, n - 1\}$ verificar

$$[a^u \neq 1 \pmod n)] \wedge [a^{2^i u} \neq -1 \pmod n) \text{ para todo o } i \in \{1, 2, \ldots, t\}]$$

dizemos que a é uma <u>testemunha</u> de n ser composto.

Teorema 16 *Para n ímpar e para um inteiro $a \in_u \{2, 3, \ldots, n - 1\}$ (escolha aleatória uniforme), a probabilidade de a ser uma testemunha da não-primalidade de n é pelo menos $3/4$.*

Este Teorema permite finalmente encontrar um algoritmo aleatorizado que se comporta da forma seguinte



O algoritmo é o seguinte:

```
Algoritmo aleatorizado de Rabin-Miller para o problema da primalidade
Dado: n, inteiro ímpar
Resultado:
   se n é primo: resposta SIM
   se n é composto: resposta incorrecta com probabilidade <= 3/4
                    (a determinar)

   Escreva n-1 na forma 2^t * u
   a = inteiro aleatório uniforme em {2,3,...,n-1}

   x = u^(n-1)  (mod n)
   se x diferente de 1:
      return COMPOSTO
   senão:
      calcule (eficientemente):
        a^u, a^(2u), a^(3u),..., a^((t-1)u)
      se algum destes valores for congruente com -1 (mod. n):
          return PRIMO    (provavelmente)
      senão:
          return COMPOSTO
```

Exercício 39 *Mostre que o algoritmo anterior pode ser implementado em tempo polinomial (em termos de* $\log n$*).*

Exercício 40 *Suponha que pretende saber se um inteiro dado é primo ou não com uma probabilidade de erro inferior a* $0.000\,000\,001$*. Quantas vezes deve correr o algoritmo anterior para garantir esse majorante do erro? Se obtiver o resultado "*`COMPOSTO`*", que pode concluir? E se obtiver o resultado "*`PRIMO`"?

4.6 Computação aleatorizada: classes de complexidade

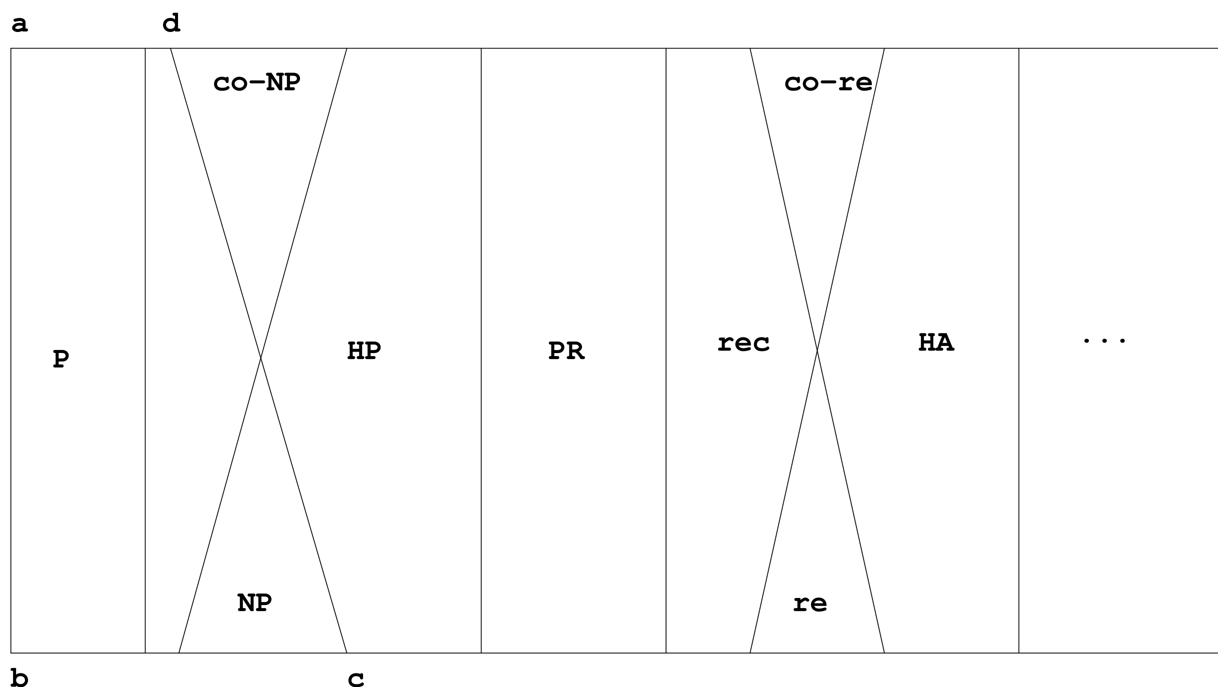
4.6.1 Panorama geral das classes de complexidade

O diagrama seguinte contém um sumário das classes de complexidade e de computabilidade dos predicados[9] Deve notar-se o seguinte

– Cada classe inclui todas as classes que estão para a esquerda; por exemplo, a classe **NP** corresponde à área do trapézio `a-b-c-d`.

– De forma implícita, o diagrama anterior define também as diferenças entre classes.

– A classe dos predicados decidíveis é a classe `rec`, a qual inclui evidentemente todos as classes à esquerda.



Legenda:

– **P** : Classe dos predicados decidíveis em tempo polinomial.

– **NP** : Classe dos predicados decidíveis em tempo polinomial por uma máquina de Turing não determinística.

– **HP** : Hierarquia polinomial, $\bigcup_{i\geq 0}(\Sigma_i^p \cup \Pi_i^p)$

– `PR`: Classe dos predicados definíveis por recursão primitiva.

[9] Quando falamos em "predicados" podemos em alternativa falar em "linguagens"; é basicamente a mesma coisa; ao predicado $p(x)$ corresponde a linguagem $L = \{x : p(x)\}$ e à linguagem L corresponde o predicado $x \in L$.

– `rec`: Classe dos predicados recursivos.

– `re`: Classe dos predicados recursivivamente enumeráveis.

– `co-re`: Classe complementar de `re`.

– **HA** : Hierarquia aritmética, $\bigcup_{i\geq 0}(\Sigma_i \cup \Pi_i)$

Este diagrama está ainda muito incompleto. Indicamos dois exemplos desse facto: (i) a classe **P** pode ser sub-dividida em sub-classes que traduzem a dificuldade de solução dos predicados polinomiais. (ii) A classe dos problemas completos em NP, ou "NP-completos", também não está indicada (pertence a $\mathbf{NP}\backslash\mathbf{P}$).

4.6.2 Classes de complexidade aleatorizadas

Vamos agora definir as classes aleatorizadas mais importantes. Vamos utilizar um esquema de definição semelhante ao usado em [7]; esse esquema inclui também as classes **P** e **NP**.

Representamos por x os dados do problema e por y uma variável aleatória que é a testemunha do algoritmo aleatorizado ou do problema **NP**. Seja $n = |x|$ e $l(n)$ uma função de n majorada por um polinómio, sendo $|y| = l(n)$. Quando escrevemos $y \in_u \{0,1\}^{l(n)}$ pretendemos dizer que y tem a distribuição uniforme (isto é, a mesma probabilidade, igual a $1/2^{l(n)}$) em $\{0,1\}^{l(n)}$.

Definição.

– **P** : Classe das linguagens L cuja função característica $f\colon \{0,1\}^n \to \{0,1\}$ é computável em tempo polinomial.

– **NP** : Classe das linguagens L tais que existe uma função computável em tempo polinomial $f\colon \{0,1\}^n \times \{0,1\}^{l(n)} \to \{0,1\}$ tal que

$$\begin{cases} x \in L & \Rightarrow \quad \text{prob}_{\,y\in_u\{0,1\}^{l(n)}}[f(x,y)=1] > 0 \\ x \notin L & \Rightarrow \quad \text{prob}_{\,y\in_u\{0,1\}^{l(n)}}[f(x,y)=1] = 0 \end{cases}$$

– **RP** : Classe das linguagens L para as quais existe uma constante $\varepsilon > 0$ e uma função computável em tempo polinomial $f\colon \{0,1\}^n \times \{0,1\}^{l(n)} \to \{0,1\}$ tais que

$$\begin{cases} x \in L & \Rightarrow \quad \text{prob}_{\,y\in_u\{0,1\}^{l(n)}}[f(x,y)=1] \geq \varepsilon \\ x \notin L & \Rightarrow \quad \text{prob}_{\,y\in_u\{0,1\}^{l(n)}}[f(x,y)=1] = 0 \end{cases}$$

– **BPP** : Classe das linguagens L para as quais existem constantes $\varepsilon > 0$ e ε' com $0 \leq \varepsilon < 1/2 < \varepsilon' \leq 1$ e uma função computável em tempo polinomial $f\colon \{0,1\}^n \times \{0,1\}^{l(n)} \to \{0,1\}$

tais que

$$\begin{cases} x \in L & \Rightarrow & \text{prob}_{\ y \in_u \{0,1\}^{l(n)}}[f(x,y)=1] \geq \varepsilon' \\ x \notin L & \Rightarrow & \text{prob}_{\ y \in_u \{0,1\}^{l(n)}}[f(x,y)=1] \leq \varepsilon \end{cases}$$

Exercício 41 *Mostre que esta definição da classe* **NP** *coincide com a definição que conhece.*

Exercício 42 *Se conhecer (apenas) o algoritmo aleatorizado de Rabin-Miller para a primalidade, o que pode dizer sobre a classe do problema da primalidade (ou da linguagem correspondente)? E, na realidade, a que classe pertence?*

Exercício 43 *Mostre que*

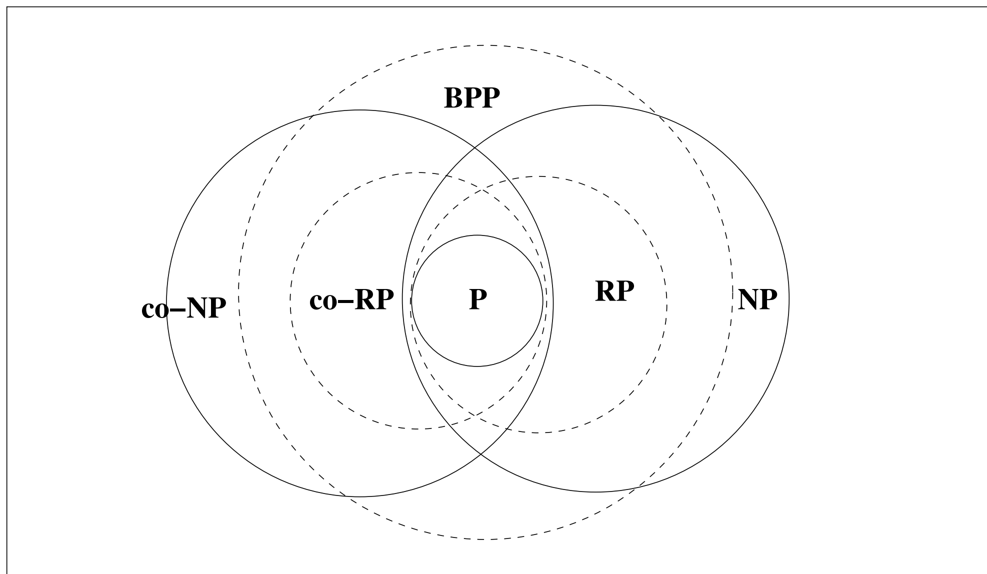
a) **RP**⊆**NP**, *co*-**RP**⊆*co*-**NP**.

a) *co*-**BPP**=**BPP**

Exercício 44 *Mostre que poderíamos ter arbitrado* $\varepsilon = 1/2$ *na definição de* **RP**. *A solução deste problema usa um algoritmo que consiste executar um número fixo um outro algoritmo.*

Exercício 45 *Mostre que poderíamos ter arbitrado* $\varepsilon = 1/3$ *e* $\varepsilon' = 2/3$ *na definição de* **BPP**.

A relação entre estas classes de complexidade esté ilustrada na figura seguinte. Note-se que não há (tanto quanto se sabe) nenhuma relação de inclusão entre **BPP**, **NP** e co-**NP**.


BPP
co−NP
co−RP
P
RP
NP