# Teste 29 Novembro 2021

## Algoritmos e Complexidade

### Universidade do Minho

**Questão 1 [12 valores]**
Considere a função `firstOdd` em baixo, que calcula o índice da *primeira ocorrência de um número ímpar* num array.

```
int firstOdd (int v[], int N) {
  //pre: N>=0
  int i=0;
  while (i<N && v[i]%2==0) i++;

  if (i==N) r = -1;

  else r = i;
  //pos: (r==-1 && forall_{0 <= k < N} ...) || (0<=r<N  && ... )
  return r;
}
```

1. Complete a definição da pós-condição da função, preenchendo os espaços `...` .

2. Apresente um invariante para o ciclo que seja adequado para provar a correcção parcial da função face à especificação acima.

3. Apresente a análise de melhor e pior caso do tempo de execução, contando o número de comparações `v[i]%2==0` efectuadas.

4. Efectue agora a análise do caso médio de execução da função, calculando o valor esperado do número de acessos ao array assumindo aleatoriedade no

preenchimento do array.

**Resolução:**

1.

```
pos: (r==-1 && forall_{0 <= k < N} v[k]%2==0) ||
     (0<=r<N  && v[r]%2==1 && forall_{0 <= k < r} v[k]%2==0))
```

2.

```
invariant: 0<=i<=N && forall_{0 <= k < i} v[k]%2==0
```

3. Melhor caso ocorre quando $v[0]\%2 == 1$ (primeiro elemento do array é ímpar), $T_m(N) = 1$

   Pior caso ocorre quando $\forall_{0 \leq k < N-1}.\ v[k]\%2 == 0$ (os N-1 primeiros elementos são pares, sendo indiferente a paridade do último)
   $T_p(N) = N$

4. Assume-se que a probabilidade de cada posição do array conter um número ímpar é $\frac{1}{2}$. Por outro lado o facto de um elemento ser par não afecta em nada a paridade dos restantes elementos do array.
   Sendo assim, a probabilidade de serem feitas $k$ comparações é $(\frac{1}{2})^{k-1} * \frac{1}{2} = (\frac{1}{2})^k$, correspondendo à situação em que os $k-1$ primeiros elementos são pares e o seguinte é ímpar.
   Resta apenas notar que existem duas situações de pior caso, igualmente prováveis, em que é feito o número máximo $N$ de comparações, pelo que acrescentaremos uma parcela final fora do somatório:

   $T(N) = \left(\sum_{k=1}^{N} (\frac{1}{2})^k * k\right) + (\frac{1}{2})^N * N$

   Da expansão
   $T(N) = \frac{1}{2} * 1 + \frac{1}{4} * 2 + \frac{1}{8} * 3...$

   intui-se que $T(N) = \theta(1)$. De facto, a probabilidade de ser feito no máximo um pequeno número de comparações é independente do comprimento do array. Por exemplo a probabilidade de serem feitas uma ou duas comparações é sempre de 0.75, por muito que façamos crescer o comprimento do array.

**Questão 2 [6 valores]**

A seguinte função `diferentes` conta de forma recursiva o número de elementos diferentes presentes num array.

```
int elem (int x, int u[], int N) {
  // pre: ...
  if (N==0) r = 0;
  else if (x==u[0]) r = 1;
       else r = elem(x, u+1, N-1));
  // pos: ...
  return r;
}

int diferentes (int v[], int N) {
  int r = 0;
  if (N!=0) {
    r = diferentes (v+1, N-1);
    if (!elem (v[0], v+1, N-1) r++;
  }
  return r;
}
```

1. Apresente uma especificação (i.e. uma pré-condição e uma pós-condição) para a função auxiliar `elem`.

2. Efectue a análise do tempo de execução da função `diferentes` no melhor e no pior caso, escrevendo e resolvendo todas as recorrências necessárias.

**Resolução:**

1.

```
pre: N >= 0
pos: (r == 1 && exists_{0 <= k < N} u[k]== x) ||
```

```
    (r == 0 && forall_{0 <= k < N} u[k]!= x)
```

Ou uma das seguintes alternativas para a pós-condição, vendo r como Booleano:

```
pos: (r  && exists_{0 <= k < N} u[k]== x) ||
     (!r && forall_{0 <= k < N} u[k]!= x)
```

```
pos: (r  -> exists_{0 <= k < N} u[k]== x) &&
     (!r -> forall_{0 <= k < N} u[k]!= x)
```

Ou ainda, de forma mais sucinta,

```
pos: r <-> exists_{0 <= k < N} u[k]== x
```

2. Contando por exemplo o número de comparações `x==u[0]` efectuadas, temos para a função `elem`:
   $T_m^{elem}(N) = 1$ (quando x está na primeira posição do array)
   $T_p^{elem}(N) = N$ (quando x não ocorre nas $N-1$ primeiras posições), resultado da recorrência:
   $T_p^{elem}(0) = 0$
   $T_p^{elem}(N) = 1 + T_p(N-1)$

   Na função `diferentes` não existe variação de comportamentos, sendo a execução sempre descrita pela recorrência:
   $T(N) = 0$ para $N \leq 1$
   $T(N) = T^{elem}(N-1) + T(N-1)$ para $N > 1$

   Note-se que para $N = 1$ a função `elem` é invocada com o array vazio (0 comparações).

   O melhor e pior caso decorrem então dos respectivos casos da função `elem`
   $T_m(N) = 1 + T(N-1)$, quando todos os elementos do array são iguais
   $T_p(N) = N - 1 + T(N-1)$, quando todos são diferentes
   (ou pelo menos os N-1 primeiros elementos)

   Expandindo e resolvendo:
   $T_m(N) = N - 1$
   $T_p(N) = \sum_{k=1}^{N-1} k = \frac{(N-1)N}{2}$

**Questão 3 [2 valores **]**

Considere a seguinte função:

```c
void loop (int v[], int N, int k) {
    int tmp, cont = 1;
    while (cont) {
        cont = 0;
        tmp = v[N-1];
        for (i=N-1; i>0; i--)
            if (v[i] != v[i-1]) {
                v[i] = v[i-1];
                if ((!cont) && (i != N-1)) cont = 1;
            }
        if (v[0]==tmp)  v[0] = (v[0]+1)%k;
    }
}
```

1. Mostre que a execução pode não terminar no caso geral, e que termina sempre se k>N (apresentando para isso um variante adequado).

2. Considerando que k>N, efectue a análise de melhor e de pior caso do tempo de execução da função.

**Resolução:**

1. Cada iteração do ciclo principal da função faz essencialmente um *shift* à direita de todo o array, com a seguinte diferença: caso o primeiro e o último elementos sejam iguais, então v[0] será incrementado mod k. A flag `cont`

garante a continuação da execução enquanto houver elementos diferentes no array (testando se pelo menos um par de adjacentes era diferente antes do shift).

Um exemplo possível (com k > 5):

```
1 2 3 1 5
1 1 2 3 1
2 1 1 2 3
2 2 1 1 2
3 2 2 1 1
3 3 2 2 1
3 3 3 2 2
3 3 3 3 2
3 3 3 3 3
```

Para mostrar que a função não termina basta apresentar uma execução que começa com um determinado array que volta a surgir ao fim de algumas iterações. Seja k = 4 na seguinte execução, em que v[0] é sempre incrementado:

```
0 3 2 1 0
1 0 3 2 1
2 1 0 3 2
3 2 1 0 3
0 3 2 1 0
```

A terminação é certa quando v[0] não ocorre no resto do array. A não-terminação implica que v[0] vá sendo incrementado, mas isto só acontecerá se existirem outras ocorrências deste valor, que entretanto sejam propagadas até à última posição. Para k>N garantidamente existirá um valor entre 0 e K-1 que não ocorre no array, e quando v[0] atingir este valor o programa terminará quando ele for propagado por todo o array.

Os pares de elementos adjacentes diferentes podem ou não diminuir em número ao ser feito um shift, mas deslocam-se sempre para a direita. Por isso o variante do ciclo principal poderá ser uma soma

$$V = \sum_{k=0}^{N-1} (N-k) * c_i$$

em que $c_i = 1$ se v[i] ≠ v[i-1], 0 em caso contrário. O multiplicador $N-k$ garante a diminuição do peso de cada par diferente quando é feito um shift.

$c_0$ terá de ser definido de forma diferente: será a distância (mod K) a que se encontra do valor de convergência $C$, que poderá ser qualquer um dos inteiros entre 0 e K-1 que não ocorrem no array. Podemos por exemplo escolher para $C$ o inteiro mais próximo (mod K) do valor inicial de v[0] que não ocorra em v. Esta distância diminuirá sempre que v[0] for incrementado.

O seguinte exemplo ilustra a evolução do variante assim definido, com $C = 4$. A necessidade de usar o multiplicador também para $c_0$ vem do facto de $c_1$ poder aumentar. quando v[0] é incrementado, como ilustrado na terceira linha:

```
1 1 2 3 1 5 -- 15+4+3+2+1 = 25
2 1 1 2 3 1 -- 15+3+2+1 = 21
3 2 1 1 2 3 -- 10+4+2+1 = 17
4 2 2 1 1 2 -- 10+3+1 = 14
5 3 2 2 1 1 -- 5+4+2 = 11
6 3 3 2 2 1 -- 5+3+1 = 9
7 3 3 3 2 2 -- 5+2 = 7
8 3 3 3 3 2 -- 5+1 = 6
9 3 3 3 3 3 -- 5
```

Note-se que 4 acabou por não ser o valor final, uma vez que o elemento 3 saiu do array, o que levou a que a convergência acontecesse com v[0]==3. Por isso o valor final do variante não foi 0.

2. No melhor caso é feito apenas um shift, em tempo linear, quando os $N-1$ primeiros elementos são já iguais e o último é diferente.

   O pior caso é quadrático. Cada execução consiste numa sequência de duas fases. Na segunda fase v[0] não ocorre no resto do array, e o algoritmo executa N shifts, cada um de tempo N, por exemplo

```
1 2 3 4 5
1 1 2 3 4
1 1 1 2 3
1 1 1 1 2
1 1 1 1 1
```

Na primeira fase o valor de v[0] vai sendo incrementado até se chegar a um valor que não ocorre no resto do array (ou ocorre apenas num segmento inicial).

```
1 2 3 1 5
1 1 2 3 1
2 1 1 2 3
2 2 1 1 2
3 2 2 1 1
(... começa fase 2)
```

Em cada shift, das duas uma: ou v[0] é incrementado, ou um elemento sai do array, diminuindo assim o número de valores diferentes pelos quais 0 pode passar, uma vez que sai um potencial trigger de incremento de 0. Como no máximo v[0] pode passar por N valores diferentes até chegar ao de convergência, não serão feitos mais de N shifts na primeira fase.

No total das duas sequências será feito portanto um número linear de shifts, cada um em tempo linear, pelo que o algoritmo executa em tempo $\mathcal{O}(N^2)$.