T2. Análise de Pior Caso, Melhor Caso, e Caso Médio

Análise de pior e melhor caso

Exemplo: o seguinte algoritmo de **procura linear** encontra a primeira ocorrência de k no array v, entre as posições a e b.

```
int procura (int v[], int a, int b, int k) {
  int i = a;
  while ((i<=b) && (v[i]!=k))
    i++;
  if (i>b)
    return -1;
  else return i;
}
```

A simples contagem do número de execuções das operações primitivas permite concluir que basta considerar como operação relevante a condição do ciclo, ou alternativamente a instrução `i++` . Consideraremos a primeira na nossa análise. Contaremos o número de *comparações* `(v[i]!=k)` efectuadas.

Ora, o número de iterações do ciclo não é agora fixo: não depende apenas do tamanho do array, mas também do seu conteúdo concreto e do valor de `k`. Seja $N = b - a + 1$.

- **Pior Caso:** quando `k` não ocorre em `v[a..b]` **OU** `k==v[b]`
  O corpo do ciclo é executado N vezes, e são feitas N **comparações**.

  Note-se que o número de comparações é o máximo (N) quer `k` não ocorra no array, ou ocorra apenas na última posição!

- **Melhor Caso:** quando `k == v[a]`, ocorre na primeira posição.
  O corpo do ciclo é executado 0 vezes, e é feita **uma comparação**.

Sendo assim, o algoritmo executa *no melhor caso em tempo constante* (0 comparações), e *no pior caso em tempo linear* (N comparações). Escreveremos:

$T(N) = \Omega(1), \mathcal{O}(N)$

EXERCÍCIO:
Repita a análise anterior contando agora as comparações `(i<=b)` efectuadas. O resultado será naturalmente o mesmo em termos assimptóticos, no entanto a contagem exacta é diferente e apresenta piores casos diferentes...

É importante perceber que esta análise de casos extremos é limitada. Em particular, não sabemos o que acontece *em média;* nem sabemos qual a probabilidade de ocorrência de cada um destes caso extremos.

No entanto, **a análise de pior caso é em geral considerada extremamente útil**, pelas seguintes razões:

- Constitui uma *garantia*, uma vez que nos dá um limite superior para o tempo de execução, válido para qualquer input do algoritmo
- Em muitos cenários o pior caso ocorre *muito frequentemente.* Por exemplo no algoritmo de pesquisa ordenada, existem certamente muitos valores de `k` que não ocorrem no array, e sempre que for feita uma pesquisa com um destes valores estaremos em presença do pior caso
- Por esta razão, o pior caso frequentemente coincide com o *caso médio*

Exemplo: algoritmo **Insertion sort**.

A função seguinte ordena o array A de forma crescente entre as posições 0 e N-1.

```
void insertionSort(int A[], int N) {
  int i, j, key;
  for (j=1 ; j<N ; j++) {
    key = A[j];
    i = j-1;
    while (i>=0 && A[i]>key) {
      A[i+1] = A[i];
      i--;
    }
    A[i+1] = key;
  }
}
```

Relembrando o funcionamento deste algoritmo, vai sendo construído um segmento inicial

ordenado do array, satisfazendo o seguinte:

> ***Invariante de ciclo:*** *no início de cada iteração do ciclo* `for`*, o vector contém entre as posições 0 e j−1 os mesmos valores que lá estavam inicialmente, já ordenados.*

Em cada passo (i.e. cada iteração do ciclo `for` exterior) é inserido no segmento ordenado um novo elemento (`key`, que está inicialmente na posição j). O ciclo interior `while` copia para a posição seguinte os elementos *superiores* ao que vai ser inserido, por forma a criar espaço para a inserção de `key`.

Compreender o funcionamento do algoritmo é importante para a análise do seu tempo de execução. Comecemos por observar que, havendo ciclos aninhados, as operações relevantes são as do *ciclo interior.* Consideraremos como operação relevante a condição `A[i]>key` .

Notemos em seguida que o número de iterações do ciclo exterior é fixo (N-1), mas o número de iterações do ciclo interior é variável.

Para uma iteração arbitrária do ciclo exterior (ou seja para um valor arbitrário de j) temos as seguintes situações extremas:

- É feita **uma comparação** `A[i]>key` que falha imediatamente.
  O elemento `key` a inserir é maior do que todos os elementos do segmento ordenado.
  O corpo do ciclo interior não é executado nenhuma vez, e `key` ficará na posição `j` em que se encontrava, uma vez que `i+1 == j` .

- É feito o número máximo de comparações: O elemento `key` a inserir é comparado com todos os elementos em `v[0..j-1]`, sendo assim feitas **`j` comparações** `A[i]>key`.
  `key` é menor do que todos os elementos do segmento ordenado, exceto possivelmente o primeiro, e será inserido numa das posições 0 ou 1, dependendo do resultado da última comparação, `A[0]>key`.

O melhor e pior caso da comparação `A[i]>key` dão-se quando estas situações acima se repetem em **todas as iterações do ciclo exterior.**

- **Melhor Caso:**
  Para que `A[i]>key` falhe imediatamente em todas as iterações do ciclo exterior, i.e. para todos os valores de `j`, é necessário que o array esteja inicialmente ***já ordenado por ordem crescente***.
  Neste caso o número total de vezes que a condição `A[i]>key)` é avaliada será então

  $$\sum_{j=1}^{N-1} 1 = N - 1.$$

- **Pior Caso:**
  Para que seja feito o número máximo **j** de comparações em todas as iterações do ciclo exterior, i.e. para todos os valores de `j`, basta que o array esteja inicialmente ***ordenado por ordem decrescente*** (inversa à pretendida).
  Neste caso o número total de vezes que a condição `A[i]>key)` é avaliada será então

  $$\sum_{j=1}^{N-1} j = \frac{(N-1)N}{2} = \frac{1}{2}N^2 - \frac{1}{2}N.$$

Sendo assim, o algoritmo executa *no melhor caso em tempo linear*, e *no pior caso em tempo quadrático.* Escreveremos:

$$T(N) = \Omega(N), \mathcal{O}(N^2)$$

> ***Notas:***
>
> *1. Não teria sido adequado optar por tomar como operação relevante (apenas) uma das atribuições no corpo do ciclo while, o que nos teria levado, incorrectamente, a um comportamento de tempo constante no melhor caso!!!*
>
> *2. O array ordenado por ordem decrescente é suficiente para que ocorra o pior caso, mas não é necessário. A caracterização mais geral do pior caso permite que uma ocorrência do menor elemento do array esteja inicialmente em qualquer posição, com todos os restantes elementos ordenados inversamente. Esta caracterização geral deve-se ao facto de ser irrelevante qual o resultado da comparação de `key` com `A[0]`.*

EXERCÍCIO: procura num array ordenado

A função seguinte pode ser utilizada para procurar a primeira ocorrência de k em v, desde que este array se encontre *ordenado* de forma crescente.

```
// requires v ordenado de forma crescente entre os índices a e b
int procuraOrd (int v[], int a, int b, int k) {
  int i = a;
  while ((i<=b) && (v[i]<k))
    i++;
  if (i>b || v[i]!= k)
    return -1;
  else return i;
}
```

Identifique o melhor e o pior caso de execução desta função e analise os respectivos tempos de execução assimptóticos. Tendo em conta essa análise como compara esta função com a função `procura` anterior?

EXERCÍCIO: algoritmo de incremento de um array de bits

Considere-se a operação de incremento de um número inteiro, representado como array de bits (*bitvector*). O primeiro 0 do array, isto é, aquele que se encontra na posição "menos significativa",

deverá passar a 1, sendo que todos os 1s que se encontram antes dele (i.e. em posições menos significativas) passarão a 0. Por exemplo:

- incrementar o bitvector 10101000 resulta em 10101001
- incrementar o bitvector 01010111 resulta em 01011000

Escreveremos o algoritmo em C considerando que os bitvectors são representados por arrays de números inteiros (com valor 0 ou 1), correspondendo o índice 0 ao bit menos significativo.

```
void inc (int b[], int N) {
  int i = 0;
  while ((i < N) && (b[i] == 1)) {
    b[i] = 0;
    i++;
  }
  if (i<N) b[i] = 1;
}
```

Identifique o melhor e o pior caso deste algoritmo (i.e. em que circunstâncias ocorrem), e analise o seu tempo de execução em cada um desses casos.

EXERCÍCIO: algoritmo de procura binária

A procura num array ordenado pode ser feita de forma mais eficiente do que anteriormente pelo seguinte algoritmo de *procura binária:*

```
// v ordenado de forma crescente entre os índices a e b
int procuraBin (int v[], int a, int b, int k) {
  int m, result := -1;
  while (a <= b && result == -1) {
    m = a + (b-a) / 2;
    if (vector[m] < k) a = m+1;
    else if (vector[m] > k) b = m-1;
         else result = m;
    }
    return result;
}
```

Analise o tempo de execução assimptótico do algoritmo no melhor e no pior caso. Diga em que situações ocorrem esses casos.

Exercícios Adicionais

1. Considere os dois seguintes algoritmos de multiplicação de números inteiros:

```
INT prod (INT x, INT y) {
    int r  = 0;
    while (y>0) {
        r = r+x;
        y = y-1;
    }
    return r;
}
INT bprod (INT x, INT y) {
    int r = 0;
    while (y>0) {
        if (y%2 != 0) r = r+x;
        x = x*2;
        y = y/2;
    }
    return r;
}
```

Como referido em +T1. Tempo de Execução de Algoritmos Iterativos, na análise deste tipo de algoritmos é importante ter em conta que o tamanho de um número não é o próprio número: a análise deve ser feita em função do comprimento da representação dos números.

Analise então o número de vezes que a operação representativa correspondente à avaliação da expressão `y>0` é executada, no melhor e no pior caso. Efectue a análise em função **do número de *bits necessários para representar*** `y`, no melhor e no pior caso, ou seja o seu tamanho útil. A análise em função do tamanho do tipo int não faz muito sentido, uma vez que este tamanho é fixo, e a análise assimptótica pressupõe a variação do tamanho.

RESOLUÇÃO

Antes de mais vejamos alguns exemplos do número de bits necessários para representar alguns números:

- para representar 2 = 10 são necessários 2 bits
- para representar 7 = 111 são necessários 3 bits
- para representar 17 = 10001 são necessários 5 bits

Colocando a questão ao contrário: quais são os números que podem ser representados com, por

exemplo, 4 bits?
Esta gama abrange os números desde 1000 = 8 até 1111 = 15.

Generalizando, com n bits é possível representar os números desde desde 2^{n-1} até $2^n - 1$.

Na função `prod` o número de avaliações de `y>0` é $y + 1$, e a variação de y entre 2^{n-1} e $2^n - 1$ permite identificar o melhor e o pior caso:

$T^{mc}(n) = 2^{n-1} + 1$, quando $y = 10 \ldots 0 = 2^{n-1}$

$T^{pc}(n) = 2^n$, quando $y = 11 \ldots 1 = 2^n - 1$

Em termos assimptóticos, o tempo de execução é em ambos os casos exponencial, $T(n) = \theta(2^n)$.

Já na função `bprod` o número de avaliações é substancialmente inferior. Considerando os números representáveis por 4 bits, temos para y=8 que y toma sucessivamente os valores 8, 4, 2, 1, 0, e para y=15 toma os valores 15, 7, 3, 1, 0. Temos então em ambos os casos 5 avaliações, e é fácil ver que em geral teremos $T(n) = n + 1 = \theta(n)$.

Uma forma de entender a diferença de funcionamento entre ambas as funções passa por observar a evolução dos números em representação binária.
Assim com `bprod` para y = 1000 temos 1000 → 100 → 10 → 1 → 0 (cada divisão por 2 corresponde a um *shift*), enquanto que com `prod` temos 1000 → 111 → 110 → 101 → 100 → 11 → 10 → 1 → 0.

---

2. Algoritmo de ordenação **Bubblesort**
   Este algoritmo, apesar de não ser considerado uma boa escolha (a análise empírica revela que de facto não é), é conceptualmente interessante. Tal como no **selection sort**, o passo básico coloca o i-ésimo elemento do vector na sua posição final, mas este passo inclui agora uma acção secundária, que vai progressivamente contribuindo para ordenar os restantes elementos do array, o que significa que pode não ser necessário efectuar N passos básicos.

```
void bubbleSort (int v[], int N){
  int i, j=0, ok=0;
  while (!ok) {
    ok = 1;
    for (i=N-1; i>j; i--)
      if (v[i-1]>v[i]) {
        swap(v, i-1, i);
        ok = 0;
      }
    j++;
  }
```

```
}
```

a. Determine o melhor e o pior caso para o número de trocas (chamadas à função `swap`).
b. Determine o melhor e o pior caso para o número de comparações entre elementos do *array*.
c. Qual das operações deverá ser considerada a relevante do ponto de vista do tempo de execução do algoritmo? Efectue a análise assimptótica do tempo de execução no melhor e no pior caso.
d. O princípio em que se baseia este algoritmo (e o invariante do ciclo exterior) é o mesmo do algoritmo *selection sort*, mas o i-ésimo elemento do vector é agora colocado na posição final i através de uma sequência de operações `swap` entre *posições adjacentes* do array. O algoritmo incorpora uma optimização que consiste em parar quando a colocação de um elemento é conseguida sem que seja necessário um único `swap`, o que significa que o vector está já ordenado.
   O algoritmo pode ser mais optimizado. Para o array `[10,20,30,40,80,70,60,50]` obtemos na primeira iteração do ciclo exterior `[10,20,30,40,50,80,70,60]`, sendo que o último `swap` efectuado nesta iteração envolveu as posições 4 (80) e 5 (50). Isto significa que os elementos nas posições $0 \ldots 4$ estão já ordenados, podendo por isso ser deixados de fora da próxima iteração.
   Implemente esta optimização e repita a análise de melhor e pior caso para o algoritmo optimizado.

3. Considere a seguinte função em C que determina se um vector de inteiros contém elementos repetidos.

```c
int repetidos (int v[], int N) {
    int i, j, rep = 0;
    for (i=0; i<N-1 && !rep; i++)
        for (j=i+1; j<N && !rep; j++)
            if (v[i]==v[j]) rep = 1;
    return rep;
}

```

a. Identifique o melhor e o pior casos da execução desta função no que respeita ao número de comparações (entre elementos do vector).
b. Quantas operações são executadas em cada caso? Efectue a análise assimptótica do tempo de execução em ambos os casos.

4. Relembre o algoritmo selection sort de +T1. Tempo de Execução de Algoritmos Iterativos:

```c
void selectionSort(int A[], int n) {
    int i, j, min, k;

```

```
    for (j=0; j < n-1; j++) {
        min = j;
        i = j+1;
        while (i < n) {
            if (A[min] > A[i]) min = i;
            i++;
        }
        if (j != min) swap (A, j, min);
    }
}
```

Claramente, a operação `swap` não é uma boa escolha para a análise assimptótica do tempo de execução. Apesar disso, proceda agora à análise do melhor e do pior caso do número de swaps executados, tendo particular cuidado com a identificação de inputs que levam ao pior caso.

Análise de caso médio

Em qualquer dos exemplos anteriores o melhor e o pior caso identificam um espectro de execuções possíveis do algoritmo, limitado inferiormente e superiormente por aqueles dois casos especiais. No entanto, a simples identificação dos casos extremos não nos fornece qualquer indicação sobre qual o comportamento do algoritmo em termos médios.

A análise de caso médio procura responder a esta questão, calculando o **valor esperado** do número de execuções das operações relevantes. Trata-se de uma noção estudada em Teoria de Probabilidades, mas que é suficientemente simples para poder ser aplicada sem grandes noções teóricas daquela área.

EXEMPLO: jogo de dados
[origem: http://www.wikihow.com/Calculate-an-Expected-

Value#Finding_the_Expected_Value_of_a_Dice_Game_sub]

Imagine-se um jogo de dados com os seguintes prémios monetários:
- 30€ caso se obtenha um 6️⃣
- 20€ caso se obtenha um 5️⃣

Cada jogada tem no entanto um custo fixo de 10€.

Para calcularmos o valor esperado de uma jogada, começamos por calcular o saldo real de cada caso (resultado de uma jogada):
- 1️⃣ -10€
- 2️⃣ -10€
- 3️⃣ -10€
- 4️⃣ -10€
- 5️⃣ 10€
- 6️⃣ 20€

Calculamos agora o valor esperado como a soma pesada destes valores, tomando como pesos as probabilidades de ocorrência dos diferentes casos. Tratando-se de um dado, a probabilidade de ocorrer cada face é a mesma: $\frac{1}{6}$. Logo a probabilidade de o saldo real ser -10€ é $4/6$; a probabilidade de ser 10€ é $1/6$, e a probabilidade de ser 20€ é também $1/6$.

Sendo assim temos:
$E = \frac{4}{6}(-10) + \frac{1}{6}10 + \frac{1}{6}20$

Note-se que *a soma das probabilidades de todos os casos é sempre igual a uma unidade.* Neste caso $\frac{4}{6} + \frac{1}{6} + \frac{1}{6} = 1$.

O valor esperado é então $E = -1.67$€, pelo que se trata de um jogo que, *em média, não compensa jogar.*

Exemplo: algoritmo de incremento de um array de bits

Recorde a operação de incremento de um número inteiro, representado como array de bits. O primeiro 0 do array, isto é, aquele que se encontra na posição "menos significativa", deverá passar a 1, sendo que todos os 1s que se encontram antes dele (i.e. em posições menos significativas) passarão a 0. Por exemplo:

- incrementar o bitvector 10101000 resulta em 10101001
- incrementar o bitvector 01010111 resulta em 01011000

```
void inc (int b[], int N) {
  int i = 0;
  while ((i < N) && (b[i] == 1)) {
    b[i] = 0;
```

```
    i++;
  }
  if (i<N) b[i] = 1;
}
```

Naturalmente, assumiremos que todos os bitvectors (inputs) possíveis ocorrem com igual probabilidade.

Consideremos como operação primitiva relevante o número de *bit flips* efectuados correspondentes a *passagens do valor de um bit de 1 a 0 ou de 0 a 1*.
Este número varia entre os casos extremos:

- 1 bit flip quando *o bit menos significativo (índice 0) é 0*, e
- N bit flips quando *os N-1 bits menos significativos têm valor 1.*

Método Baseado em Contagem

> *Para calcular o valor esperado do número de bit flips teremos que escrever a soma dos vários casos possíveis, pesados pela respectiva probabilidade. Para isso efectuamos uma contagem dos inputs correspondentes a cada um dos casos:*
>
> *- metade dos bitvectors de comprimento N têm o bit menos significativo 0;*
> *- dos restantes, metade têm o segundo bit menos significativo 0, i.e. terminam em 01;*
> *- dos restantes, metade têm o segundo bit menos significativo 0, i.e. terminam em 001;*
> *- e assim sucessivamente.*
> *Temos então*
>
> $$T(N) = \mathbf{\frac{1}{2}} * 1 \text{ flip} + \mathbf{\frac{1}{4}} * 2 \text{ flips} + \mathbf{\frac{1}{8}} * 3 \text{ flips} + \ldots + \mathbf{2 * \frac{1}{2^N} * N}$$
>
> *A probabilidade* $\mathbf{2 * \frac{1}{2^N}}$ *de ocorrerem* $\mathbf{N}$ *bit flips corresponde a duas situações que podem ocorrer, ambas com probabilidade* $\mathbf{\frac{1}{2^N}}$ *:*
> *– a situação em que todos os bits são 1 excepto o mais significativo, e*
> *– a situação em que todos são 1.*
> *Por exemplo, com 4 bits, haverá 4 bit flips quer com 0111 → 1111 quer com 1111 → 0000.*
>
> $$T(N) = \sum_{k=1}^{N} \mathbf{\frac{1}{2^k} * k} + \mathbf{\frac{1}{2^N} * N}$$
>
> *E logo, uma vez que* $\sum_{k=1}^{\infty} k/2^k = 2$*,*
> $T(N) < 2 = \mathcal{O}(1)$
>
> *Trata-se pois de um algoritmo cujo comportamento é, no caso médio, assimptoticamente igual ao comportamento no melhor caso.*
>
> ***Nota:*** *o caso médio de* $T(N)$ *tende para o valor 2, o que não significa que 2 seja o número de bitflips executado* com maior frequência — *esse valor é 1.*

Método baseado em eventos e probabilidade condicionada

> *Em alternativa a este método baseado em contagem, podemos alternativamente calcular a probabilidade $\frac{1}{2^k}$ de serem executados k bit flips, considerando os eventos que terão de ocorrer para que isso aconteça, e tendo em conta que a noção de conjunção lógica se traduz pela multiplicação de probabilidades envolvendo um condicionamento, da seguinte forma:*
>
> *probabilidade de os $k-1$* bits menos significativos serem 1 **e** o bit seguinte ser 0
> *=*
> probabilidade de o bit $k-1$ ser 0, *condicionada* ao facto de os $k-1$ menos significativos serem 1
> *
> probabilidade de os $k-1$ bits menos significativos serem 1
>
> $P(\ b[0] = 1 \wedge \ldots \wedge b[k-2] = 1 \wedge b[k-1] = 0\ )$
> $=$
> $P(\ b[k-1] = 0 \ \mid \ b[0] = 1 \wedge \ldots \wedge b[k-2] = 1\ ) * P(\ b[0] = 1 \wedge \ldots \wedge b[k-2] = 1\ )$
>
> *Ora, o efeito do condicionamento é aqui irrelevante (os valores dos bits são todos independentes): a probabilidade de um qualquer bit ser 0 ou 1 é sempre de $\frac{1}{2}$.*
>
> *Temos então a probabilidade:*
>
> $P(\ b[k-1] = 0\ ) * P(\ b[0] = 1 \wedge \ldots \wedge b[k-2] = 1\ )$
> $=$
> $P(\ b[k-1] = 0\ ) * P(\ b[k-2] = 1\ ) * \ldots * P(\ b[0] = 1\ )$
>
> $\frac{1}{2} * (\frac{1}{2})^{k-1} = \frac{1}{2^k}$
>
> *Este cálculo pressupõe que é feito o bitflip 0 → 1. O caso adicional de k=N com todos os bits a 1 tem também de ser considerado, resultando como anteriormente no caso médio:*
>
> $T(N) = \sum_{k=1}^{N} \frac{1}{2^k} * \mathbf{k} + \frac{1}{2^N} * \mathbf{N}$

Exemplo: algoritmo de procura linear

O exemplo que acabamos de ver ilustra uma situação pouco comum, em que o caso médio do tempo de execução coincide com o melhor caso. Vejamos agora um exemplo de uma situação

mais comum, em que isso não se verifica.

Relembremos o algoritmo de pesquisa linear. O que pretendemos agora é calcular o valor esperado da operação que foi considerada relevante para a análise de melhor e pior caso (a comparação `v[i]!=k` .

```
int procura (int v[], int a, int b, int k) {
  int i = a;
  while ((i<=b) && (v[i]!=k))
    i++;
  if (i>b)
    return -1;
  else return i;
}
```

Método baseado em eventos e probabilidade condicionada

Ponto Prévio

Se se tratar de um array de números inteiros de m bits, existirão 2^m números diferentes, e assumindo aleatoriedade temos que:

- a probabilidade de ocorrência de um qualquer número numa posição do array é $\frac{1}{2^m}$
- a probabilidade de **não-**ocorrência de um qualquer número numa posição do array é $1 - \frac{1}{2^m}$
- A probabilidade de um número não ocorrer num conjunto de j posições de um array é dada por

  $$(1 - \frac{1}{2^m})^j$$

  Trata-se de uma simples multiplicação, uma vez que a ocorrência numa posição não condiciona a ocorrência em qualquer outra posição

Caso Geral

> *Para $i \leq N$, caso em que k ocorre no array, teremos a seguinte* ***probabilidade de serem feitas i comparações:***
>
> probabilidade de k não ocorrer nas primeiras $i - 1$ posições
> *
> probabilidade de k ocorrer na posição i, *condicionada* pelo facto de não ocorrer nas primeiras $i - 1$ posições.

Ora, o efeito do condicionamento é novamente aqui irrelevante (os valores guardados nas diversas posições do array são todos independentes), pelo que temos a seguinte probabilidade:

$(1 - \frac{1}{2^m})^{i-1} * \frac{1}{2^m}$

Quanto ao caso em que são feitas $\mathbf{N}$ *comparações porque* k *não ocorre no array, temos probabilidade* $(1 - \frac{1}{2^m})^N$ *de ocorrência deste cenário.*

Escrevemos então o valor esperado do tempo de execução como uma soma dos vários casos possíveis, pesados pela respectiva probabilidade:

$$T(N) = \left(\sum_{i=1}^{N} (1 - \frac{1}{2^m})^{i-1} * \frac{1}{2^m} * \mathbf{i}\right) + (1 - \frac{1}{2^m})^N * \mathbf{N}$$

em que o segundo termo da soma corresponde à situação em que são feitas N *comparações porque k não ocorre em todo o array. Note-se que sem este termo, a soma das probabilidades consideradas não seria exactamente unitária.*

Para valores razoavelmente grandes de m*, facilmente se vê que* $T(N) \approx N$*, uma vez que a probabilidade de ocorrência de um qualquer valor numa determinada posição do array é praticamente nula: o caso médio é igual ao pior caso.*

Caso particular: assuma-se que k ocorre de certeza em v[ ]

Qual será o comportamento médio do algoritmo, tendo em conta apenas os casos em que o elemento procurado ***ocorre*** *no array?*

Antes de mais note-se que o comportamento do algoritmo é exactamente o mesmo caso k ocorra exactamente uma vez ou mais do que uma vez. Assumiremos que ocorre exactamente uma, o que permite simplificar a análise.

Continuamos a ter:

probabilidade de k não ocorrer nas primeiras $i - 1$ posições
*
probabilidade de k ocorrer na posição i, condicionada pelo facto de não ocorrer nas primeiras $i - 1$ posições.

Neste cenário em que k ocorre de certeza no array ***exactamente*** *uma vez, existem* N *possibilidades diferentes de isso acontecer. A probabilidade de k ocorrer numa posição arbitrária é* $\frac{1}{N}$*, e de ocorrer num qualquer segmento do array de comprimento* l *é* $\frac{l}{N}$*.*

Então a probabilidade de k não ocorrer nas $i - 1$ *primeiras posições é igual à probabilidade de ocorrer nas* $N - (i - 1)$ *restantes, ou seja* $1 - \frac{i-1}{N} = \frac{N-i+1}{N}$*.*

Por outro lado, ***o condicionamento agora é importante****: o facto de* k *não ocorrer nas* $i - 1$ *primeiras posições implica que ocorra necessariamente nas* $N - i + 1$ *últimas posições. A probabilidade condicionada de ocorrer na posição* i *é então* $\frac{1}{N-i+1}$*, pelo que a probabilidade de*

> *serem feitas i comparações é dada por*
>
> $\frac{N-i+1}{N} * \frac{1}{N-i+1} = \frac{1}{N}$
>
> *E temos então que o valor esperado de $T(N)$ é:*
>
> $T(N) = \frac{1}{N}1 \ + \ \frac{1}{N}2 \ + \ \ldots \ + \ \frac{1}{N}N$
>
> $T(N) = \sum_{i=1}^{N} \frac{1}{N} * \mathbf{i} = \frac{N(N+1)}{2N} = \frac{N+1}{2}$
>
> *Este resultado está de acordo com o que poderíamos intuitivamente esperar: se k ocorre com igual probabilidade em qualquer posição do array, então* em média *o número de comparações feitas será o correspondente à situação em que k ocorre no meio do array.*

Método Baseado em Contagem

Aplicaremos este método apenas ao caso particular em que **k ocorre exactamente uma vez no array v.**

> *Uma vez que k pode ocorrer* com igual probabilidade em qualquer posição do array*, a probabilidade de esta ocorrência ser numa determinada posição i é de $\frac{1}{N}$, qualquer que seja i.*
>
> *Neste caso a condição* `(i<=b)` *será sempre verdadeira, uma vez que o ciclo terminará garantidamente antes de i ultrapassar o limite superior do array. Basta pois contar o número de comparações* `(v[i]!=k)` *entre k e elementos do array.*
>
> *Contemos então o número de comparações* `(v[i]!=k)` *efectuadas, dependendo da posição em que k ocorre pela primeira vez:*
> *- 1 comparação se k ocorre na posição a*
> *- 2 comparações se k ocorre na posição a+1*
> *...*
> *- N comparações se k ocorre na posição b*
>
> *O* número esperado de comparações *pode agora ser calculado escrevendo a soma pesada destes números, sendo o peso a probabilidade de cada caso, ou seja:*
>
> $\frac{1}{N}1 \ + \ \frac{1}{N}2 \ + \ \ldots \ + \ \frac{1}{N}N$
>
> $= \frac{1}{N} \sum_{i=1}^{N} i = \frac{N(N+1)}{2N} = \frac{N+1}{2}$

EXERCÍCIO: algoritmo **Insertion sort**

```
void insertionSort(int A[], int N) {
  int i, j, key;
  for (j=1 ; j<N ; j++) {
    key = A[j];
    i = j-1;
    while (i>=0 && A[i]>key) {
      A[i+1] = A[i];
      i--;
    }
    A[i+1] = key;
  }
}
```

RESOLUÇÃO

Relembremos: o número de iterações do ciclo exterior é fixo (N-1), e para cada valor de j, o número de vezes que a condição `(i>=0 )` do ciclo interior é avaliada (que é a operação considerada anteriormente para a análise) pode variar entre 1 e j+1. Se escrevermos n_j para designar este número, o número de operações relevantes efectuadas pode ser escrito como

$$T(N) = \sum_{j=1}^{N-1} n_j, \text{ com } 1 \leq n_j \leq j+1.$$

Para calcularmos o *valor esperado* deste número, temos mais uma vez que assumir *aleatoriedade* no preenchimento do array, o que implicará que a probabilidade de a inserção ser feita em qualquer posição do segmento já ordenado do *array* é **a mesma**. Temos $j+1$ situações diferentes, pelo que, por contagem, o valor desta probabilidade é $\frac{1}{j+1}$.

Sendo assim, o valor esperado do número de vezes que a condição `i>=0` é avaliada, para um determinado valor de j, é

$$n_j = \sum_{k=1}^{j+1} \frac{1}{j+1} k = \frac{1}{j+1} \sum_{k=1}^{j+1} k = \frac{1}{j+1} \frac{(j+1)(j+2)}{2} = \frac{j}{2} + 1.$$

Note-se que este valor esperado a que chegamos vai de encontro à seguinte intuição:

> *Uma vez que assumimos que o array foi preenchido aleatoriamente, então em cada iteração do ciclo exterior, de entre os elementos do segmento ordenado do array (entre as posições 0 e j-1), metade dos elementos é superior a* `A[j]`.

Ou seja, "em média", a inserção será feita a meio do segmento já ordenado do *array*.

Considerando agora o tempo global de execução do algoritmo, temos assim
$T(N) = \sum_{j=1}^{N-1} (\frac{j}{2} + 1)$

$T(N) = \frac{1}{2} \frac{(N-1)N}{2} + N - 1 = \frac{1}{4}N^2 + \frac{3}{4}N - 1 = \Theta(N^2)$

O comportamento no caso médio é assimptoticamente quadrático (tal como no pior caso). Trata-se de uma conclusão importante, uma vez que o comportamento de melhor caso (linear) parece muito promissor — mas na realidade, *em média*, o comportamento do *insertion sort* não é melhor do que o do *selection sort*.

EXERCÍCIO: algoritmo de procura num array ordenado

```
// requires v ordenado de forma crescente entre os índices 0 e N-1
int procuraOrd (int v[], int N, int k) {
  int i = 0;
  while ((i<N) && (v[i]<k))
    i++;
  if (i>N-1 || v[i]!= k)
    return -1;
  else return i;
}
```

Calcule o caso médio do tempo de execução deste algoritmo, estimando o valor esperado do número de comparações (<). Para isso, assuma que o array ordenado está preenchido de forma aleatória com valores dentro de uma dada gama, e que k é escolhido aleatoriamente dentro dessa gama.