Aula 5 - Análise da Complexidade de Algoritmos

**\*\*\* Entregue, num ficheiro ZIP, este guião preenchido e o código desenvolvido \*\*\***

**1 –** Considere uma sequência (*array*) de **n** **valores reais**. Pretende-se determinar se os elementos da sequência são sucessivos termos de uma **progressão geométrica**:

**r = a[ 1 ] / a[ 0 ]** e **a[ i ] = r × a[ i – 1 ], i > 1.**

* Implemente uma função **eficiente** (utilize um algoritmo em lógica negativa) e **eficaz** que verifique se os n elementos (n > 2) de uma sequência de valores reais são sucessivos termos de uma progressão geométrica. A função deverá devolver 1 ou 0, consoante a sequência verificar ou não essa propriedade.

**Depois de validar o algoritmo apresente a função no verso da folha.**

* Pretende-se determinar experimentalmente a **ordem de** **complexidade do número de multiplicações e divisões** efetuadas pelo algoritmo e envolvendo elementos da sequência.
* Considere as seguintes sequências de 10 elementos, que cobrem as distintas situações possíveis de execução do algoritmo. Determine, para cada uma delas, se satisfazem a propriedade e qual o número de operações de multiplicação e de divisão efetuadas pelo algoritmo.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |  | Resultado | 0 |  | Nº de operações | 2 |
| 1 | 2 | 4 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |  | Resultado | 0 |  | Nº de operações | 3 |
| 1 | 2 | 4 | 8 | 5 | 6 | 7 | 8 | 9 | 10 |  | Resultado | 0 |  | Nº de operações | 4 |
| 1 | 2 | 4 | 8 | 16 | 6 | 7 | 8 | 9 | 10 |  | Resultado | 0 |  | Nº de operações | 5 |
| 1 | 2 | 4 | 8 | 16 | 32 | 7 | 8 | 9 | 10 |  | Resultado | 0 |  | Nº de operações | 6 |
| 1 | 2 | 4 | 8 | 16 | 32 | 64 | 8 | 9 | 10 |  | Resultado | 0 |  | Nº de operações | 7 |
| 1 | 2 | 4 | 8 | 16 | 32 | 64 | 128 | 9 | 10 |  | Resultado | 0 |  | Nº de operações | 8 |
| 1 | 2 | 4 | 8 | 16 | 32 | 64 | 128 | 256 | 10 |  | Resultado | 0 |  | Nº de operações | 9 |
| 1 | 2 | 4 | 8 | 16 | 32 | 64 | 128 | 256 | 512 |  | Resultado | 1 |  | Nº de operações | 9 |

**Depois da execução do algoritmo responda às seguintes questões:**

* Qual é a sequência (ou as sequências) que corresponde(m) ao melhor caso do algoritmo?

|  |
| --- |
| A sequência que corresponde ao melhor caso do algoritmo é o primeiro array, porque o 3 elemento é não verifica logo a primeira condição que é arr[3] = arr[2] \* r |

* Qual é a sequência (ou as sequências) que corresponde(m) ao pior caso do algoritmo?

|  |
| --- |
| A sequência que corresponde ao pior caso é a o último array porque é a sequência que chega verifica todos os elementos e por isso até ao penúltimo elemento a sequência é uma progressão geométrica. |

* Determine o número de operações efetuadas no caso médio do algoritmo (**para n = 10**).

|  |
| --- |
| Admitindo que as sequências dadas são equiprováveis podemos obter um caso médio de operações A(n) = (2 + 3 + 4 + 5 + 6 +7 + 8 + 9 + 9) / 9 = 5,8(8) que é aproximadamente ≃ 6 operações. |

* Qual é a ordem de complexidade do algoritmo?

|  |
| --- |
| Olhando experimentalmente e admitindo que as sequências são equiprováveis podemos olhar para o pior caso e verificamos que W(n) = size – 1, onde size é o tamanho do array, daí tiramos que a ordem de complexidade é O(n). Assim fazemos uma espécie de majoração para a complexidade. |

* **Determine formalmente a ordem de complexidade do algoritmo nas situações do melhor caso, do pior caso e do caso médio, considerando uma sequência de tamanho n.** Deve obter expressões matemáticas exatas e simplificadas. **Faça essas análises no verso da folha.**

Função

|  |
| --- |
| int func(double a[], int size)  {      assert(size > 2 && "The array must have 3 elements or more.");      double r = a[1]/a[0];      nOps++;      for (int i = 2; i < size; i++)      {          nOps++;          if (a[i] != r \* a[i-1]) return 0;      }      return 1;  } |

Análise Formal do Algoritmo

|  |
| --- |
| **Nota:** É de notar que é adicionado + 1 aos somatórios porque para todas as sequências válidas é feita uma operação para calcular a razão, uma única vez antes de iniciar o algoritmo.  No Worst Case há uma vertente estática no nosso algoritmo no sentido em que faz uma comparação para cada elemento do array. Todos os elementos são comparados logo, podemos traduzir isso neste somatório:  O Best Case acontece quando o primeiro elemento a ser verificado não verifica a condição de progressão geométrica, logo apenas fazemos 2 comparações.  No Average Case, consideramos que as sequências dadas são equiprováveis daí os valores possíveis estarem a ser multiplicados por 1/(n-1). Depois dividimos os casos em 2 partes, a 1ª mostra a tendência dos primeiros n-2 elementos em que o nº de operações depende de i+1, e depois o 2º caso é o último elemento (n-1) que repete o valor anterior obtido (n-1) |

* Calcule o valor das expressões para n = 10 e **compare-os com os resultados obtidos experimentalmente**.

|  |
| --- |
| Para o W(n) se usarmos a expressão para n = 10, ficamos com W(n) = 9, quando comparado com os dados obtidos o pior número de comparações também é 9.  Para o B(n) se usarmos a expressão para n = 10, ficamos com B(n) = 2, quando comparado com os dados obtidos o melhor número de comparações também é 2.  Para o A(n) se usarmos a expressão para n = 10, ficamos com A(n) = 5,8(8), quando comparado com os dados obtidos o caso médio também dá 5,8(8) que arredondamos para 6 operações. |

**2 –** Considere uma sequência (array), possivelmente não ordenada, de n elementos inteiros e positivos. Pretende-se **eliminar os elementos da sequência que sejam iguais ou múltiplos ou submúltiplos de algum dos seus predecessores**, sem fazer a sua ordenação e sem alterar a posição relativa dos elementos.

Por exemplo, a sequência { 2, 2, 2, 3, 3, 4, 5, 8, 8, 9 } com 10 elementos será transformada na sequência { 2, 3, 5 } com 3 elementos; e a sequência { 7, 8, 2, 2, 3, 3, 3, 8, 8, 9 } com 10 elementos será transformada na sequência { 7, 8, 3, } com 3 elementos.

* Implemente uma função **eficiente** e **eficaz** que elimina os elementos iguais ou múltiplos ou submúltiplos de algum dos seus predecessores numa sequência com n elementos (n > 1). **A função deverá ser *void* e alterar o valor do parâmetro indicador do número de elementos efetivamente armazenados na sequência (que deve ser passado por referência)**.

**Depois de validar o algoritmo apresente a função no verso da folha.**

* Pretende-se determinar experimentalmente a **ordem de** **complexidade do número de comparações** e **do número de deslocamentos** (i.e., cópias) efetuados pelo algoritmo e envolvendo elementos da sequência.
* Considere as sequências anteriormente indicadas de 10 elementos e outras à sua escolha. Determine, para cada uma delas, a sua configuração final, bem como o número de comparações e de deslocamentos efetuados.

**Depois da execução do algoritmo responda às seguintes questões:**

* Indique uma sequência inicial com 10 elementos que conduza ao **melhor caso do número de comparações** efetuadas. Qual é a sequência final obtida? Qual é o número de comparações efetuadas? Qual é o número de deslocamentos (i.e., cópias) de elementos efetuados? Justifique.

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Inicial: | 1 | X | X | X | X | X | X | X | X | X | Nº de comparações | 18 |
| Final: | 1 | / | / | / | / | / | / | / | / | / | Nº de cópias | 0 |

|  |
| --- |
| As sequências que coincidem com o melhor caso do número de comparações coincidem com todas as sequências que começam com o valor 1, os elementos restantes não interessam todos os elementos iram ser sempre múltiplos, divisores ou iguais a 1, fazendo com que o número de comparações e cópias seja estático, 18 e 0, respetivamente.  Outro tipo de sequência que tem o mesmo comportamento são todas as aquelas em que os elementos são todos iguais entre si.  **Nota**: X é qualquer número >= 1, e ‘/’ significa nenhum elemento. |

* Indique uma sequência inicial com 10 elementos que conduza ao **pior caso do número de comparações** efetuadas. Qual é a sequência final obtida? Qual é o número de comparações efetuadas? Qual é o número de deslocamentos (i.e., cópias) de elementos efetuados? Justifique.

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Inicial: | 2 | 3 | 5 | 7 | 11 | 13 | 17 | 19 | 23 | 29 | Nº de comparações | 108 |
| Final: | 2 | 3 | 5 | 7 | 11 | 13 | 17 | 19 | 23 | 29 | Nº de cópias | 0 |

|  |
| --- |
| As sequências que coincidem com o pior caso para comparações são aquelas que contém apenas elementos primos e os dois primeiros elementos são iguais (ou o segundo é múltiplo ou divisor do primeiro). Quando isto acontece o primeiro elemento é repetido logo é retirado e depois todos os elementos seguintes têm que ser deslocados. |

* Determine formalmente a ordem de complexidade do algoritmo nas situações do **melhor caso** e do **pior caso**, considerando uma sequência de tamanho n. Deve obter expressões matemáticas exatas e simplificadas. **Faça essas análises no verso da folha.**

**Função**

|  |
| --- |
| void func(int a[], int\* size)  {      assert(\*size > 1 && "The array must have 2 elements or more.");        int cont = 1;  // Posição onde se escreve valores válidos      for (int i = 1; i < \*size; i++)      {          nComp++;          if(isValid(a[i], a, cont))          {                  nComp++;                  // Não rescrever valores que já estão no sitio correto                  if(a[cont] != a[i])                  {                      nCopy++;  // Rescrever array com valor válido                      a[cont++] = a[i];                  } else                      cont++;          }      }      \*size = cont;  }  int isValid(int element, int a[], int i)  {      for (int j = 0; j < i; j++)      {          nComp++;  // Verificar se o elemento é multiplo ou igual a a[j]          if(element % a[j] == 0) return 0;          nComp++;  // Verificar se o elemento é divisor de a[j]          if(a[j] % element == 0) return 0;      }      return 1;  } |

Análise Formal do Algoritmo – Comparações – Melhor Caso – Pior Caso

|  |
| --- |
| No Worst Case todos os elementos são primos logo nenhum deles é múltiplo, igual ou divisor entre eles, logo vai haver 2i comparações dentro da função *isValid()* mais 2 comparações que são sempre feitas dentro do primeiro ciclo for na função *func()*. Daí retiramos a seguinte expressão:  No Best Case só fazemos uma comparação dentro da função *func()* porque nenhum dos restantes elementos é válido. Dentro da função *isValid()* uma comparação é válida porque o elemento vai ser sempre múltiplo ou igual a a[j]. Daí retiramos a seguinte expressão: |

Análise Formal do Algoritmo – Deslocamentos – Melhor Caso – Pior Caso

|  |
| --- |
| No Worst Case o segundo elemento do array é igual, múltiplo ou divisor do primeiro e os restantes dos elementos são números primos, assim todos os elementos depois do 2 elemento vão ter que ser deslocados.  Daí resulta a seguinte expressão:  No Best Case várias coisas podem acontecer como o primeiro elemento ser 1, todos os elementos serem iguais e todos os elementos serem números primos (não repetidos).  Quando isto acontece nenhum elemento tem que ser copiado, porque apenas o primeiro é que é válido e os restantes não vão fazer parte do array final. |