**Análise de algoritmos**

**Aplicação e análise assintótica de algoritmos**

**Herlander Francisco Sanguluvulo da Silva;**

Ciências da computação – Centro Universitário do Triangulo (UNI -TRI) – AV- Nicomedes Alves dos Santos, 4545, Bairro Gávea – CEP 38.411-106 – Uberlândia – MG – Brasil

[herlanderfranciscomandela@hotmail.com](mailto:herlanderfranciscomandela@hotmail.com)

**Resumo**: Este trabalho visa a implementação de uma análise de desempenho de algoritmos baseada no tempo, bem como o comportamento de ordenação aplicado para vetores de tamanhos distintos, fazendo uso das amostras e dados encontrados para gerar executar cálculos estatísticos e entender as diferenças existentes entre as várias aplicações.

Abstract: This work aims the implementation of a time-based algorithm performance analysis, as well as the ordering behavior applied to vectors of different sizes, using the samples and data found to generate statistical calculations and understand the differences between various applications.

**1.Introdução**

**1.1 - Análise de algoritmo**

Em ciência da computação, a análise de algoritmos tem como função determinar os recursos necessários para executar um dado algoritmo. A maior parte dos algoritmos são pensados para trabalhar com entradas (inputs) de tamanho arbitrário. Em geral, a eficiência ou complexidade de um algoritmo é função do tamanho do problema, do número de passos necessário (complexidade temporal) e da complexidade espacial ou de memória do sistema usado para executar o algoritmo. Esta disciplina faz parte da mais vasta teoria da complexidade computacional, que permite fazer estimativas quanto aos recursos necessários para que um algoritmo resolva um determinado problema computacional.

Assim, o objetivo final não é apenas fazer códigos que funcionem, mas que sejam também eficientes. Para isso, deve-se estudar alguns tipos de problemas que podem ser resolvidos computacionalmente. Em seguida, deve ser visto como a abordagem adotada para resolver pode influenciar, levando a um algoritmo mais ou menos eficiente.

"Ao verificar que um dado programa está muito lento, uma pessoa prática pede uma máquina mais rápida ao seu chefe. Mas o ganho potencial que uma máquina mais rápida pode proporcionar é tipicamente limitado por um fator de 10, por razões técnicas ou econômicas. Para obter um ganho maior, é preciso buscar melhores algoritmos. Um bom algoritmo, mesmo rodando em uma máquina lenta, sempre acaba derrotando (para instâncias grandes do problema) um algoritmo ruim rodando em uma máquina rápida. Sempre."

**1.2 - Algoritmos e tempo de execução**

Um algoritmo é uma sequência finita de instruções bem definidas e não ambíguas, cada uma das quais devendo ser executadas mecânica ou eletronicamente em um intervalo de tempo finito e com uma quantidade de esforço finita.

A sua aplicação pode ser composta por uma porção de algoritmos, cada um destinado a um fim muito específico. Por exemplo, você pode ter um algoritmo responsável por encontrar todos os pedidos vendidos no último mês, que contenham um determinado produto. Com o advento do Big Data, inúmeros algoritmos são postos em prática para mineração e análise de dados, então, mesmo que exista uma aplicação ou serviço resolvendo esses problemas para você, acredite… os algoritmos estarão lá.

Um determinado algoritmo pode ter tempos de execução relativamente diferentes de acordo com o ambiente no qual ele esteja rodando. Se for num computador Core i7 e 16GB de RAM, é possível assumirmos que ele rodará consideravelmente melhor do que se estivesse operando em um Raspberry Pi, por exemplo. Ainda há um segundo cenário onde, talvez você tenha escrito o algoritmo perfeito em Python ou Ruby, mas ele corre o risco de executar de forma mais lenta que um algoritmo em Assembly ou C.

Partindo da premissa que um bom algoritmo é um conjunto de operações que resolvem um problema em tempo e esforço atrativos, surge a necessidade de o classificarmos como sendo rápido ou não, dando assim espaço a análise assintótica.

**1.3 - Análise assintótica**

Segundo o Wikibooks, a análise assintótica é “(…) a way of expressing the main component of the cost of an algorithm, using idealized (not comparable) units of computational work ”.

Em termos mais práticos, é uma forma de julgarmos se o nosso algoritmo é eficiente, independente dos “recursos que o cercam” (como velocidade de processamento, quantidade de memória, latência de rede, etc).

Removendo todas as variáveis que podem influenciar no tempo de execução, focamos nossas atenções em como o algoritmo está escrito, em qual é a sua entrada, e se “ele por si” é a maneira mais eficiente para a resolução de um determinado problema.

Vale reforçar que a entrada é um fator de extrema importância no que tange a análise assintótica. A análise é “input bound”, ou seja, a entrada influenciará diretamente no resultado do estudo. Por exemplo, quando ordenamos um vetor de tamanho N, utilizando o algoritmo Selection Sort, teremos um tempo de execução de N2 (já que o algoritmo pega um número, e compara com os demais números no vetor, repetindo essa operação até chegar ao fim do dado estruturado).

Ao fim da análise, podemos chegar a 2 conclusões diferentes: Melhor cenário e pior cenário.

**1.4 - Big O, Big Omega e Big Theta**

Estas são notações assintóticas muito famosas na análise de tempos de execução de algoritmos.

* O(n): Expressa o limite superior do tempo de execução de um algoritmo (pior cenário);
* Ω(n): Expressa o limite inferior do tempo de execução de um algoritmo (melhor cenário);
* Θ(n): Expressa limite superior e inferior do tempo de execução de um algoritmo (pior e melhor cenário).

Além da expressão linear, temos outras notações que descrevem diferentes tempos de execução:

* O(1): Constante
* O(log n): Logarítmica
* O(n): Linear
* O(n log n): “Linearithmic” (maior que linear, menor que quadrática)
* O(n2): Quadrática
* O(n3): Cúbica
* nO(1): Polinomial
* 2O(n): Exponencial

De maneira simplista, N pode ser considerado como o número de operações que o algoritmo leva para chegar ao seu final. N está intimamente ligado com a entrada do seu algoritmo, onde quanto maior for o seu número, maior será o seu tempo de execução.

1.5 **Algoritmos de Ordenação**

**1.5.1 - InsertionSort**

Sua teoria baseia-se em ordenar os valores da esquerda para a direita, deixando os elementos lidos (a esquerda) ordenados. Este é geralmente utilizado para ordenar um pequeno número de valores, onde faz-se muito eficiente. A complexidade do código é:

Complexidade Pior Caso: O(n²)

Complexidade Caso Médio: O(n²)

Complexidade Melhor Caso: O(n)

Quando temos um caso onde a complexidade é n² devemos evitar, visto que a redução de desempenho deste algoritmo é exponencial. Porém, no seu melhor caso temos uma constante n que significa a inalteração da velocidade, proporcional à quantidade de elementos.

Lembre-se que estamos trabalhando com proporcionalidade, então dizer que não varia não significa que um vetor de 10 elementos será ordenado na mesma velocidade de um vetor de um milhão de elementos, mesmo no melhor caso, porém a proporcionalidade entre a quantidade de elementos e sua velocidade continua constante, diferente do Pior Caso que aumenta ao quadrado.

O melhor caso ocorre quando todos os elementos já estão ordenados e o pior caso é exatamente o contrário, quando todos os elementos estão desordenados.

**1.5.2 - Aplicando o Insertion sort:**

|  |
| --- |
| public static void main(String[] args) throws IOException {  int quantidade = 10000;  int[] vetor = new int[quantidade];  for (int i = 0; i < vetor.length; i++) {  vetor[i] = (int) (Math.random()\*quantidade);  }  long tempoInicial = System.currentTimeMillis();  insertionSort(vetor);  long tempoFinal = System.currentTimeMillis();  System.out.println("Executado em = " + (tempoFinal - tempoInicial) + " ms");  }  public static void insertionSort(int[] vetor) {  int j;  int key;  int i;  for (j = 1; j < vetor.length; j++)  {  key = vetor[j];  for (i = j - 1; (i >= 0) && (vetor[i] > key); i--)  {  vetor[i + 1] = vetor[i];  }  vetor[i + 1] = key;  }  } |

**Análise assintótica**

|  |  |  |
| --- | --- | --- |
| **InsertionSort** | | |
| **Elementos** | Caso médio (n²) | Melhor Caso (n) |
| 10.000 | 220 ms | 1 ms |
| 100.000 | 5110 ms | 6 ms |
| 1.000.000 | 151498 ms | 15 ms |
| 10.000.000 | ms | Ms |
| 100.000.000 | ms | Ms |

**Tabela 1**

**Análise gráfica insertion sort.**

**Figura 1**

**BubbleSort**

O BubbleSort é conhecido pela sua simplicidade e pela eficácia ao realizar ordenações em um número limitado de valores. Seu princípio baseia-se na troca de valores entre posições consecutivas, fazendo com que valores altos ou baixos (dependendo da forma de ordenação desejada) “borbulhem” para o final da fila, por isso este algoritmo é chamado de BubbleSort. Sua complexidade é:

Complexidade Pior Caso: O(n²)

Complexidade Caso Médio: O(n²)

Complexidade Melhor Caso: O(n)

Vimos que no melhor caso o seu tempo é quase inalterável, permanecendo constante, ou seja, um caso ideal.

**Aplicando o Bubble sort:**

|  |
| --- |
| public static void main(String[] args) throws IOException {  int quantidade = 10000;  int[] vetor = new int[quantidade];  for (int i = 0; i < vetor.length; i++) {  vetor[i] = (int) (Math.random()\*quantidade);  }  long tempoInicial = System.currentTimeMillis();  bubbleSort(vetor);  long tempoFinal = System.currentTimeMillis();  System.out.println("Executado em = " + (tempoFinal - tempoInicial) + " ms");  }  private static void bubbleSort(int vetor[]){  boolean troca = true;  int aux;  while (troca) {  troca = false;  for (int i = 0; i < vetor.length - 1; i++) {  if (vetor[i] > vetor[i + 1]) {  aux = vetor[i];  vetor[i] = vetor[i + 1];  vetor[i + 1] = aux;  troca = true;  }  }  }  } |

**Análise assintótica**

|  |  |  |
| --- | --- | --- |
| **BubbleSort** | | |
| **Elementos** | Caso médio (n²) | Melhor Caso (n) |
| 10.000 | 200 ms | 0 ms |
| 100.000 | 20.426 ms | 2 ms |
| 1.000.000 | 6222320ms | 8 Ms |
| 10.000.000 | ms | Ms |
| 100.000.000 | ms | Ms |

**Tabela 2**

**Análise gráfica Bubble sort:**

**Figura 2**

***QuickSort***

Este algoritmo usa uma técnica conhecida por divisão e conquista, muito conhecida por reduzir problemas complexos em problemas menores para tentar chegar em uma solução. Sendo assim, o resultado do problema inicial é dada como a soma do resultado de todos os problemas menores. Sua complexidade é:

* Complexidade Pior Caso: O(n²)
* Complexidade Caso Médio: (nlogn)
* Complexidade Melhor Caso: (nlogn)

O QuickSort sai na frente de outros algoritmos mais simples quando tratamos do caso médio, por trabalhar com logaritmo (nlogn), o que torna o resultado mais rápido do que o InsertionSort e o QuickSort.

**Aplicando o Quick sort:**

|  |
| --- |
| public static void main(String[] args) throws IOException {  int quantidade = 10000;  int[] vetor = new int[quantidade];  for (int i = 0; i < vetor.length; i++) {  vetor[i] = (int) (Math.random()\*quantidade);  }  long tempoInicial = System.currentTimeMillis();  quickSort(vetor,0,vetor.length-1);  long tempoFinal = System.currentTimeMillis();  System.out.println("Executado em = " + (tempoFinal - tempoInicial) + " ms");  }  private static void quickSort(int[] vetor, int inicio, int fim) {  if (inicio < fim) {  int posicaoPivo = separar(vetor, inicio, fim);  quickSort(vetor, inicio, posicaoPivo - 1);  quickSort(vetor, posicaoPivo + 1, fim);  }  private static int separar(int[] vetor, int inicio, int fim) {  int pivo = vetor[inicio];  int i = inicio + 1, f = fim;      while (i <= f) {  if (vetor[i] <= pivo)  i++;  else if (pivo < vetor[f])  f--;  else {  int troca = vetor[i];  vetor[i] = vetor[f];  vetor[f] = troca;  i++;  f--;  }  }  vetor[inicio] = vetor[f];  vetor[f] = pivo;  return f;  } |

**Análise assintótica**

|  |  |
| --- | --- |
| **QuickSort** | |
| **Elementos** | Caso médio (nlogn) |
| 10.000 | 39 ms |
| 100.000 | 43 ms |
| 1.000.000 | 52 ms |
| 10.000.000 | ms |
| 100.000.000 | ms |

Tabela 3

**Análise gráfica Quick sort**

**Figura 3**

**Merge Sort**

Sua ideia básica consiste em Dividir (o problema em vários sub-problemas e resolver esses sub-problemas através da recursividade) e Conquistar (após todos os sub-problemas terem sido resolvidos ocorre a conquista que é a união das resoluções dos sub-problemas).

Como o algoritmo Merge Sort usa a recursividade, há um alto consumo de memória e tempo de execução, tornando esta técnica não muito eficiente em alguns problemas.

**Aplicando o Merge sort**

|  |  |
| --- | --- |
| public class MergeSort {  public static int[] sort(int[] array) {  if (array.length <= 1) {  return array;  }  int meio = array.length / 2;  int[] dir = array.length % 2 == 0 ? new int[meio] : new int[meio + 1];  int[] esq = new int[meio];  int[] aux = new int[array.length];  for (int i = 0; i < meio; i++) {  esq[i] = array[i];  }  int auxIndex = 0;  for (int i = meio; i < array.length; i++) {  dir[auxIndex] = array[i];  auxIndex++;  }  esq = sort(esq);  dir = sort(dir);  aux = mergesort(esq, dir);  return aux;  }  static int[] mergesort(int[] esq, int[] dir) {  int[] aux = new int[esq.length + dir.length]; | |
| int indexDir = 0, indexEsq = 0, indexAux = 0;    while (indexEsq < esq.length || indexDir < dir.length) {  if (indexEsq < esq.length && indexDir < dir.length) {  if (esq[indexEsq] <= dir[indexDir]) {  aux[indexAux] = esq[indexEsq];  indexAux++;  indexEsq++;  } else {  aux[indexAux] = dir[indexDir];  indexAux++;  indexDir++;  }  } else if (indexEsq < esq.length) {  aux[indexAux] = esq[indexEsq];  indexAux++;  indexEsq++;  } else if (indexDir < dir.length) {  aux[indexAux] = dir[indexDir];  indexAux++;  indexDir++;  }  }  return aux;  }  } | |

**Heap Sort**

O Heapsort não é um algoritmo de ordenação estável. Porém, é possível adaptar a estrutura a ser ordenada de forma a tornar a ordenação estável. Cada elemento da estrutura adaptada deve ficar no formato de um par (elemento original, índice original). Assim, caso dois elementos sejam iguais, o desempate ocorrerá pelo índice na estrutura original.

**Aplicando o Heap sort:**

|  |
| --- |
| public class HeapSort {  public static int[] sort(int[] v) {  buildMaxHeap(v);  int n = v.length;  for (int i = v.length - 1; i > 0; i--) {  swap(v, i, 0);  maxHeapify(v, 0, --n);  }  return v;  }  private static void buildMaxHeap(int[] v) {  for (int i = v.length / 2 - 1; i >= 0; i--) {  maxHeapify(v, i, v.length);  }    }  private static void maxHeapify(int[] vetor, int pos, int tamanhoDoVetor) {  int max = 2 \* pos + 1, right = max + 1;  if (max < tamanhoDoVetor) {  if (right < tamanhoDoVetor && vetor[max] < vetor[right]) {  max = right;  } |

|  |
| --- |
| if (vetor[max] > vetor[pos]) {  swap(vetor, max, pos);  maxHeapify(vetor, max, tamanhoDoVetor);  }  }  }    public static void swap(int[] v, int j, int aposJ) {  int aux = v[j];  v[j] = v[aposJ];  v[aposJ] = aux;  }  } |

**Radix sort**

O Radix sort é um algoritmo de ordenação rápido e estável que pode ser usado para ordenar itens que estão identificados por chaves únicas. Cada chave é uma cadeia de caracteres ou número, e o radix sort ordena estas chaves em qualquer ordem relacionada com a lexicografia.

Na ciência da computação, radix sort é um algoritmo de ordenação que ordena inteiros processando dígitos individuais. Como os inteiros podem representar strings compostas de caracteres (como nomes ou datas) e pontos flutuantes especialmente formatados, radix sort não é limitado somente a inteiros.

|  |
| --- |
| public class TestRadix {  private static final int MAX\_CHARS = 28;  private static void radixSort(String[] v) {  Queue<String> queues[] = createQueues();  for (int pos = maxSize(v) - 1; pos >= 0; pos--) {  for (int i = 0; i < v.length; i++) {  int q = queueNo(v[i], pos);  queues[q].add(v[i]);  }  restore(queues, v);  }  }  private static void restore(Queue<String>[] qs, String[] v) {  int contv = 0;  for (int q = 0; q < qs.length; q++)  while (qs[q].size() > 0)  v[contv++] = qs[q].remove();  }  private static Queue<String>[] createQueues() {  LinkedList<String>[] result = new LinkedList[MAX\_CHARS];  for (int i = 0; i < MAX\_CHARS; i++) {  result[i] = new LinkedList<String>();  }  return result;  }  private static int queueNo(String string, int pos) {  if (pos >= string.length()) {  return 0;  } |
| char ch = string.charAt(pos);  if ((ch >= 'A') && (ch <= 'Z')) {  return (ch - 'A' + 1);  }  else if (ch >= 'a' && ch <= 'z') {  return ch - 'a' + 1;  }  else {  return 27;  }  }  private static int maxSize(String[] v) {  int maxValue = v[0].length();  for (int i = 1; i < v.length; i++) {  if (maxValue < v[i].length()) {  maxValue = v[i].length();  }  }  return maxValue;  }  public static void printStringArray(String[] arrToPrint) {  for (int i = 0; i < arrToPrint.length; i++) {  System.out.print(arrToPrint[i]+" ");  }  System.out.println();  } | | |

|  |
| --- |
| public static void main(String[] args) {  System.out.print("Input: ");  printStringArray(args);  radixSort(args);  System.out.print("\nOutput: ");  printStringArray(args);  }  }  public void radixSort(int vector[]){  for(int digit = 0; digit < 3; digit ++){  int power = (int) Math.pow(10, digit+1);  int z[][] = new int[vector.length][10];  int n[] = new int[10];  for(int i = 0; i < vector.length; i ++){  int num = vector[i];  z[n[(num%power)/(power/10)]][(num%power)/(power/10)] = num;  n[(num%power)/(power/10)]++;  }  int c = 0;  for(int i = 0; i < 10; i ++){  for(int j = 0; j < vector.length; j ++){  if(j < n[i]){  vector[c] = z[j][i];  c++;  }else{break;}  }  }    }  } |

**Bucket Sort**

Bucket sort, ou bin sort, é um algoritmo de ordenação que funciona dividindo um vetor em um número finito de recipientes. Cada recipiente é então ordenado individualmente, seja usando um algoritmo de ordenação diferente, ou usando o algoritmo bucket sort recursivamente. O Bucket Sort tem complexidade linear Ꝋ(n) quando o vetor a ser ordenado contém valores que são uniformemente distribuídos.

**Aplicado o Bucket sort:**

|  |
| --- |
| public static void BucketSort(int[] vetor, int maiorValor)  {  int numBaldes = maiorValor/5;  LinkedList[] B = new LinkedList[numBaldes];  for (int i = 0; i < numBaldes; i++){  B[i] = new LinkedList<Integer>();  }  //Coloca os valores no balde respectivo:  for (int i = 0; i < vetor.length; i++) {  int j = numBaldes-1;  while (true){  if(j<0){  break;  }  if(vetor[i] >= (j\*5)){  B[j].add(vetor[i]);  break;  }  j--;  }  } |

|  |
| --- |
| //Ordena e atualiza o vetor:  int indice = 0;  for (int i = 0; i < numBaldes; i++){  int[] aux = new int[B[i].size()];  //Coloca cada balde num vetor:  for (int j = 0; j < aux.length; j++){  aux[j] = (Integer)B[i].get(j);  }  insertionSort(aux); //algoritmo escolhido para ordenação.  // Devolve os valores ao vetor de entrada:  for (int j = 0; j < aux.length; j++, indice++){  vetor[indice] = aux[j];  }  }  } |

**Especificação da máquina utilizada para testar os softwares:**

**Sistema operacional**: Windows 10

**Processador:** Intel(R) core(TM) I5 5200U CPU @ 2.20 GH

**Tipo de sistema:** 64 bits.

Bibliografia

<https://www.devmedia.com.br/algoritmos-de-ordenacao>

<https://www.geeksforgeeks.org/radix-sort>

<https://pt.wikipedia.org/wiki/Radix_sort>

<https://pt.wikipedia.org/wiki/Bucket_sort>

<https://github.com/kelvins/Algoritmos-e-Estruturas-de-Dados/blob/master/Java/HeapSort.java>