ALGORITMOS EM INFORMÁTICA

Pedro Ferreira, Pedro Guerra

Versão 1

ALGORITMOS EM INFORMÁTICA

DETI

Pedro Ferreira, Pedro Guerra
(98620) pedrodsf@ua.pt, (98610) pedroguerra@ua.pt

19/12/2021

Conteúdo

Capítulo 1

Introdução

1.1 O que é um Algoritmo?

Um algoritmo pode ser descrito como a solução de um dado problema, a partir de um conjunto de ações/operações realizadas numa determinada ordem.

Algo como uma receita de culinária, o ato de ligar o carro, ou fazer uma chamada são exemplos da utilização de algoritmos no quotidiano.

1.2 Algoritmos em Informática

Algoritmos são especialmente importantes na área de informática, visto que todos os programas, quer sejam de alto ou baixo nível, se regem por **conjuntos finitos de instruções bem definidas**, ou seja, algoritmos.

Capítulo 2

Metodologia

O conteúdo e organização deste documento foi baseada no livro **"Análise de Complexidade de Algoritmos"** [1].

O código apresentado para cada algoritmo em anexo foi retirado do site **https://www.geeksforgeeks.org/**, com os respetivos autores devidamente identificados.

Capítulo 3

Complexidade de Algoritmos

A medida dos recursos computacionais temporais ou espaciais (memória) consumidos por um algoritmo é chamada de **complexidade algorítmica**.

É de extrema relevância classificar a complexidade de cada algoritmo, para assim os classificar em termos de eficiência. Para tal, é importante ter em conta:

- O melhor caso (Best Case Efficiency)
- O pior caso (Worst Case Efficiency)
- O caso médio (Average Case Efficiency)

Neste documento damos especial atenção aos recursos computacionais temporais, visto que grande parte dos algoritmos apresentados utiliza pouca memória auxiliar (apenas para as variáveis necessárias, não afetando de forma considerável o seu desempenho).

Sabendo que os recursos temporais utilizados por cada algoritmo (tempo de execução) diferem significativamente com a máquina que os executa, não é possível comparar diretamente as respetivas eficiências. Por tal, tem-se em conta a taxa de crescimento do tempo de execução em função do número de elementos de entrada. Com isto, é possível obter uma equação ao interpolar o valor obtido para cada número de elementos de entrada.

Para classificar o algoritmo, tem-se em conta a variação do tempo de execução da equação referida anteriormente e não os valores específicos obtidos. Tendo como objetivo a classificação da eficiência, apenas se analisa o limite superior de crescimento da função (**Big O Notation**).

Podemos ter as seguintes ordens de complexidade algorítmica:

Ordem	Complexidade
$O(1)$	Constante
$O(log_2 n)$	Logarítmica
$O(n)$	Linear
$O(nlog_2 n)$	Linear-logarítmica
$O(n^c)$	Polinomial
$O(c^n)$	Exponencial
$O(n!)$	Fatorial

Tabela 3.1: Ordens de Complexidade Algorítmica

Capítulo 4

Algoritmos de Pesquisa

Para algo como a pesquisa de um elemento numa lista, são utilizados algoritmos de pesquisa. No caso da lista não ser ordenada, apenas poderemos realizar uma pesquisa linear (Sequential Search), caso contrário poderemos usar outras estratégias, tais como uma pesquisa binária (Binary Search) ou ternária (Ternary Search)

4.1 Sequential Search

Este algoritmo realiza uma pesquisa exaustiva (**Força Bruta**), isto é, itera sobre todos os elementos, desde o início ao fim, até encontrar o que procura.
Este algoritmo tem complexidade $O(n)$.

4.2 Binary Search

Numa sequência ordenada, a pesquisa binária utiliza a estratégia decrementar para conquistar (**Decrease and Conquer**) com o fator de 2, logo divide a sequência a meio e seleciona a metade que contém o elemento pretendido, dividindo de seguida a metade selecionada. Este processo acaba quando o elemento for encontrado, ou até não ser possível dividir o conjunto em dois.

Este algoritmo tem complexidade $O(log_2 n)$.

4.3 Ternary Search

Utiliza o a mesma estratégia que a pesquisa binária, mas usando um fator de 3, ou seja, divide a sequência em três partes e seleciona a que contém o elemento pretendido, dividindo de seguida a parte selecionada.

Este algoritmo tem complexidade $O(log_3 n)$.

Capítulo 5

Algoritmos de Ordenação

Para rearranjar elementos de acordo com uma comparação (crescente, decrescente) são usados os algoritmos de ordenação. Estes podem ser **estáveis**, quando preservam a ordem relativa dos elementos repetidos na sequência, ou **instáveis**. Também podem ser classificados por **ordenação interna**, quando os elementos se encontram armazenados numa sequência, ou **ordenação externa**, quando os mesmos estão armazenados em ficheiros.

5.1 Sequential Sort

Tal como na pesquisa sequencial, a ordenação sequencial é realizada uma pesquisa de força bruta, iterando sobre cada elemento e comparando-o com os restantes. Quando o elemento a comparar for maior que o elemento comparado, ocorre uma troca de posição.

Este algoritmo tem complexidade $O(n^2)$.

5.2 Selection Sort

A ordenação seleção consiste em comparar cada elemento aos restantes da sequência. Caso existam valores menores, o menor valor trocará de posição com o valor da posição atual.

Este algoritmo tem complexidade $O(n^2)$.

5.3 Bubble Sort

O algoritmo bubble sort itera sobre a sequência e compara o valor da posição atual com o da posição seguinte. Se for maior, os valores trocam de posição. Isto faz com que, ao fim de iterar sobre a lista, o último valor fique ordenado, pelo que já não será preciso iterar até à posição final (possível otimização).

Este algoritmo tem complexidade $O(n^2)$.

5.4 Insertion Sort

Tal como na pesquisa binária e ternária, é usada a estratégia decrementar para conquistar, sendo colocados os valores um a um na sequência, de forma ordenada.

Este algoritmo tem complexidade $O(n^2)$.

5.5 Shell Sort

Tendo por base o insertion sort, o shell sort permite ordenar os valores que distam um salto entre si. Após estarem ordenados, o salto diminui e o ciclo é executado novamente. A sequência estará ordenada quando o salto for 1.

Este algoritmo tem complexidade $O(n^{7/6})$.

5.6 Heap Sort

O algoritmo cria uma árvore binária com os elementos da sequência, em que os nós criança são menores que o nó pai (Binary Heap).

Este algoritmo tem complexidade $O(log_2 n)$.

5.7 Merge Sort

Utilizando a estratégia dividir para conquistar (**Divide and Conquer**), o merge sort divide continuamente segmentos em duas metades até obter segmentos unitários. Por último, junta todos os segmentos até formar um só organizado (Merge).

Este algoritmo tem complexidade $O(nlog_2 n)$.

5.8 Insertion Recursive Sort

O insertion recursive sort funciona da mesma maneira que a ordenação de cartas num jogo de cartas. Para cada iteração, executa-se o algoritmos pelos valores já organizados, incluíndo o valor da posição atual.

Este algoritmo tem complexidade $O(n^2)$.

5.9 Quick Sort

Baseado no merge sort, o quick sort em vez de dividir a sequência em metade, utiliza pivôs (valores da sequência) para a divisão e ordenação de cada segmento.

Este algoritmo tem complexidade $O(nlog_2 n)$.

Contribuições dos autores

PF - Introdução

PF - Complexidade de Algoritmos

PF - Algoritmos de Pesquisa/Seleção

PF - Algoritmos de Ordenação

Bibliografia

[1] António Adrego da Rocha, *Análise da Complexidade de Algoritmos*. fev. de 2014.

Capítulo 6

Apêndices

Apêndice 1 - Sequential Search

```python
def search(arr, n, x):

    for i in range(0, n):
        if (arr[i] == x):
            return i
    return -1



# Driver Code
arr = [2, 3, 4, 10, 40]
x = 10
n = len(arr)

# Function call
result = search(arr, n, x)
if(result == -1):
    print("Element is not present in array")
else:
    print("Element is present at index", result)
```

Apêndice 2 - Binary Search

```python
def binarySearch(arr, l, r, x):

    # Check base case
    if r >= l:

        mid = l + (r - l) // 2

        # If element is present at the middle itself
        if arr[mid] == x:
            return mid

        # If element is smaller than mid, then it
        # can only be present in left subarray
        elif arr[mid] > x:
            return binarySearch(arr, l, mid-1, x)

        # Else the element can only be present
        # in right subarray
        else:
            return binarySearch(arr, mid + 1, r, x)

    else:
        # Element is not present in the array
        return -1


# Driver Code
arr = [2, 3, 4, 10, 40]
x = 10

# Function call
result = binarySearch(arr, 0, len(arr)-1, x)

if result != -1:
    print("Element is present at index % d" % result)
else:
    print("Element is not present in array")
```

Apêndice 3 - Ternary Search

```python
def binarySearch(arr, l, r, x):

    # Check base case
    if r >= l:

        mid = l + (r - l) // 2

        # If element is present at the middle itself
        if arr[mid] == x:
            return mid

        # If element is smaller than mid, then it
        # can only be present in left subarray
        elif arr[mid] > x:
            return binarySearch(arr, l, mid-1, x)

        # Else the element can only be present
        # in right subarray
        else:
            return binarySearch(arr, mid + 1, r, x)

    else:
        # Element is not present in the array
        return -1


# Driver Code
arr = [2, 3, 4, 10, 40]
x = 10

# Function call
result = binarySearch(arr, 0, len(arr)-1, x)

if result != -1:
    print("Element is present at index % d" % result)
else:
    print("Element is not present in array")
```

Apêndice 4 - Sequential Sort

```python
# Traverse through all array elements
for i in range(len(A)):

    # Find smaller elements in remaining
    # unsorted array
    for j in range(i+1, len(A)):
        # Swap the found smaller element with
        # the first element
        if A[i] > A[j]:
            A[i], A[j] = A[j], A[i]



# Driver code to test above
print ("Sorted array")
for i in range(len(A)):
    print("%d" %A[i]),
```

Apêndice 5 - Selection Sort

```python
# Traverse through all array elements
for i in range(len(A)):

    # Find the minimum element in remaining
    # unsorted array
    min_idx = i
    for j in range(i+1, len(A)):
        if A[min_idx] > A[j]:
            min_idx = j

    # Swap the found minimum element with
    # the first element
    A[i], A[min_idx] = A[min_idx], A[i]

# Driver code to test above
print ("Sorted array")
for i in range(len(A)):
    print("%d" %A[i]),
```

Apêndice 6 - Bubble Sort

```
def bubbleSort(arr):
    n = len(arr)

    # Traverse through all array elements
    for i in range(n):

        # Last i elements are already in place
        for j in range(0, n-i-1):

            # traverse the array from 0 to n-i-1
            # Swap if the element found is greater
            # than the next element
            if arr[j] > arr[j+1] :
                arr[j], arr[j+1] = arr[j+1], arr[j]

# Driver code to test above
arr = [64, 34, 25, 12, 22, 11, 90]

bubbleSort(arr)

print ("Sorted array is:")
for i in range(len(arr)):
    print ("%d" %arr[i]),
```

Apêndice 7 - Insertion Sort

```
def insertionSort(arr):

    # Traverse through 1 to len(arr)
    for i in range(1, len(arr)):

        key = arr[i]

        # Move elements of arr[0..i-1], that are
        # greater than key, to one position ahead
        # of their current position
        j = i-1
        while j >= 0 and key < arr[j] :
                arr[j + 1] = arr[j]
                j -= 1
        arr[j + 1] = key


# Driver code to test above
arr = [12, 11, 13, 5, 6]
insertionSort(arr)
for i in range(len(arr)):
    print ("% d" % arr[i])

# This code is contributed by Mohit Kumra
```

Apêndice 8 - Shell Sort

```python
def shellSort(arr):
    gap = len(arr) // 2 # initialize the gap

    while gap > 0:
        i = 0
        j = gap

        # check the array in from left to right
        # till the last possible index of j
        while j < len(arr):

            if arr[i] >arr[j]:
                arr[i],arr[j] = arr[j],arr[i]

            i += 1
            j += 1

            # now, we look back from ith index to the left
            # we swap the values which are not in the right order.
            k = i
            while k - gap > -1:

                if arr[k - gap] > arr[k]:
                    arr[k-gap],arr[k] = arr[k],arr[k-gap]
                k -= 1

        gap //= 2


# driver to check the code
arr2 = [12, 34, 54, 2, 3]
print("input array:",arr2)

shellSort(arr2)
print("sorted array",arr2)

# This code is contributed by Shubham Prashar (SirPrashar)
```

Apêndice 9 - Heap Sort

```python
# Python program for implementation of heap Sort

# To heapify subtree rooted at index i.
# n is size of heap


def heapify(arr, n, i):
    largest = i  # Initialize largest as root
    l = 2 * i + 1     # left = 2*i + 1
    r = 2 * i + 2     # right = 2*i + 2

    # See if left child of root exists and is
    # greater than root
    if l < n and arr[largest] < arr[l]:
        largest = l

    # See if right child of root exists and is
    # greater than root
    if r < n and arr[largest] < arr[r]:
        largest = r

    # Change root, if needed
    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]  # swap

        # Heapify the root.
        heapify(arr, n, largest)

# The main function to sort an array of given size


def heapSort(arr):
    n = len(arr)

    # Build a maxheap.
    for i in range(n//2 - 1, -1, -1):
        heapify(arr, n, i)

    # One by one extract elements
    for i in range(n-1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]  # swap
        heapify(arr, i, 0)


# Driver code
```

```
arr = [12, 11, 13, 5, 6, 7]
heapSort(arr)
n = len(arr)
print("Sorted array is")
for i in range(n):
    print("%d" % arr[i]),
# This code is contributed by Mohit Kumra
```

Apêndice 10 - Merge Sort

```
# Python program for implementation of MergeSort
def mergeSort(arr):
    if len(arr) > 1:

         # Finding the mid of the array
        mid = len(arr)//2

        # Dividing the array elements
        L = arr[:mid]

        # into 2 halves
        R = arr[mid:]

        # Sorting the first half
        mergeSort(L)

        # Sorting the second half
        mergeSort(R)

        i = j = k = 0

        # Copy data to temp arrays L[] and R[]
        while i < len(L) and j < len(R):
            if L[i] < R[j]:
                arr[k] = L[i]
                i += 1
            else:
                arr[k] = R[j]
                j += 1
            k += 1

        # Checking if any element was left
        while i < len(L):
            arr[k] = L[i]
            i += 1
            k += 1

        while j < len(R):
            arr[k] = R[j]
            j += 1
            k += 1

# Code to print the list
```

```python
def printList(arr):
    for i in range(len(arr)):
        print(arr[i], end=" ")
    print()


# Driver Code
if __name__ == '__main__':
    arr = [12, 11, 13, 5, 6, 7]
    print("Given array is", end="\n")
    printList(arr)
    mergeSort(arr)
    print("Sorted array is: ", end="\n")
    printList(arr)

# This code is contributed by Mayank Khanna
```

Apêndice 11 - Insertion Recursive Sort

```
def insertionSortRecursive(arr,n):
    # base case
    if n<=1:
        return

    # Sort first n-1 elements
    insertionSortRecursive(arr,n-1)
    '''Insert last element at its correct position
        in sorted array.'''
    last = arr[n-1]
    j = n-2

      # Move elements of arr[0..i-1], that are
      # greater than key, to one position ahead
      # of their current position
    while (j>=0 and arr[j]>last):
        arr[j+1] = arr[j]
        j = j-1

    arr[j+1]=last

# A utility function to print an array of size n
def printArray(arr,n):
    for i in range(n):
        print arr[i],

# Driver program to test insertion sort
arr = [12,11,13,5,6]
n = len(arr)
insertionSortRecursive(arr, n)
printArray(arr, n)

# Contributed by Harsh Valecha
```

Apêndice 12 - Quick Sort

```python
# This Function handles sorting part of quick sort
# start and end points to first and last element of
# an array respectively
def partition(start, end, array):

    # Initializing pivot's index to start
    pivot_index = start
    pivot = array[pivot_index]

    # This loop runs till start pointer crosses
    # end pointer, and when it does we swap the
    # pivot with element on end pointer
    while start < end:

        # Increment the start pointer till it finds an
        # element greater than  pivot
        while start < len(array) and array[start] <= pivot:
            start += 1

        # Decrement the end pointer till it finds an
        # element less than pivot
        while array[end] > pivot:
            end -= 1

        # If start and end have not crossed each other,
        # swap the numbers on start and end
        if(start < end):
            array[start], array[end] = array[end], array[start]

    # Swap pivot element with element on end pointer.
    # This puts pivot on its correct sorted place.
    array[end], array[pivot_index] = array[pivot_index], array[end]

    # Returning end pointer to divide the array into 2
    return end

# The main function that implements QuickSort
def quick_sort(start, end, array):

    if (start < end):

        # p is partitioning index, array[p]
        # is at right place
        p = partition(start, end, array)

```

```python
        # Sort elements before partition
        # and after partition
        quick_sort(start, p - 1, array)
        quick_sort(p + 1, end, array)

# Driver code
array = [ 10, 7, 8, 9, 1, 5 ]
quick_sort(0, len(array) - 1, array)

print(f'Sorted array: {array}')

# This code is contributed by Adnan Aliakbar
```