5/27/2019

CSP 2348 Assignment 2

A Mini Team Project

Samuel Morris – 10456767

Riegard Lemmer - 10424383

Sam

ECU JOONDALUP – CSP2348 – Data structures – assignment 2

Table of Contents

[Executive Summary 1](#_Toc9430874)

[Introduction 3](#_Toc9430875)

[Bubble and Heap Sort Algorithm Analysis 3](#_Toc9430876)

[Bubble sort algorithm analysis 3](#_Toc9430877)

[Heap sort algorithm analysis 3](#_Toc9430878)

[Summary of Sort Algorithm Complexities 4](#_Toc9430879)

[Algorithm Analysis by Experimental Studies 5](#_Toc9430880)

[Sequencing Array Sorting Algorithms Based on their Complexity 5](#_Toc9430881)

[Bubble Sort 5](#_Toc9430882)

[Selection Sort 5](#_Toc9430883)

[Insertion Sort 5](#_Toc9430884)

[. Merge Sort 5](#_Toc9430885)

# Introduction

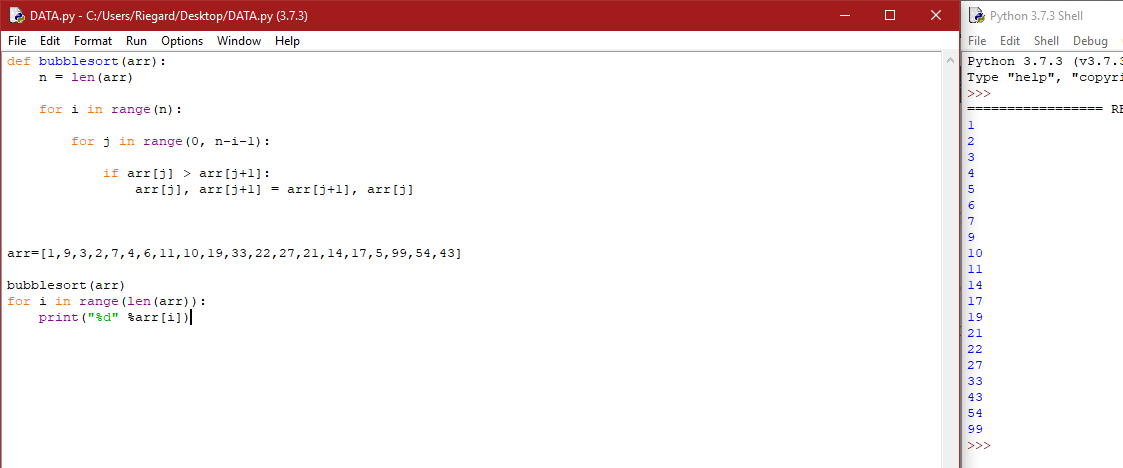
A sorting algorithm is employed to re-arrange any given array or list of elements to improve organization and efficiency. The way in which the array or elements are organised is based on the algorithm; a procedure that sorts the data in a given amount of time. The difference between sorting algorithms are their methods of sorting and efficiencies. O-notation is used to measure algorithm efficiency. It can be calculated based on the number of comparisons or run time of the algorithm (Nazeer and Sebastian 2009). This report will explore theoretical and practical analysis of sorting algorithms through design, implementation and testing. The set of sorting algorithms will include Bubble, Heap, Insertion, Selection, Merge and quick sort. These shall be implemented though arrays, singly linked lists and binary tree data structures.

# 

# Bubble and Heap Sort Algorithm Analysis

## Bubble sort algorithm analysis

bubbleSort(A : list of sortable items)

N = length of items

Repeat

Swapped = false

For i = 1 to n-1 inclusive do O(n) values in array

If A[i-1] > A[i] then

swap(A[i-1], A[i]) O(n) Operation on array

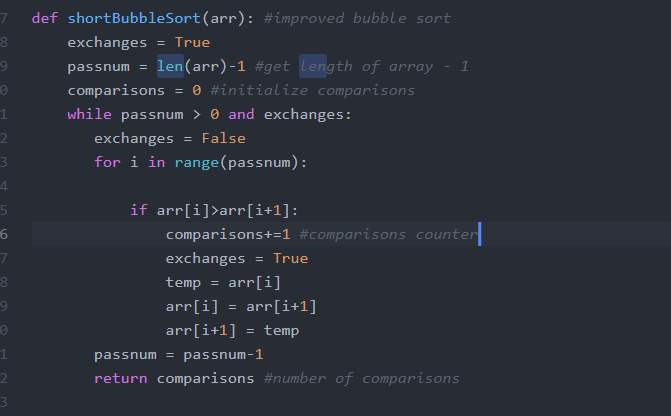
Swapped = true

End if

End for

Until not swapped

End. Overall leading running time of O(n^2)

Optimised Bubble sort

Bubblesortshot(array: List of elements)

Swapped = true

Variable = length of array(arr) - 1

Comparison counter = 0

While the length of the array is greater than 0

set swapped to false

For the value “i” in the whole array O(n) values in array

If the variable i in the array is greater than variable i +1

Count comparison counter +1

Set exchange to true

Set a variable to the current value in the array

Make the current position of i = to the next position O(n)swap

Set the next position to the temporary variable

Size of array - 1

Return the comparisons

O(N^2)

## 

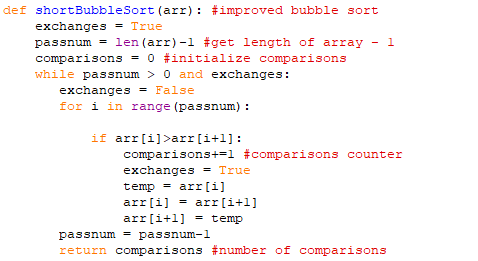
## Heap sort algorithm analysis

BuildHeap(A as array)

n = elements\_in(A)

for i = floor(n/2) to 1

Heapify(A,i,n)

Heapify(A as array, i as int, n as int)

left = 2i

right = 2i+1 O ( log n) comes from halving the lists into sublists of 1

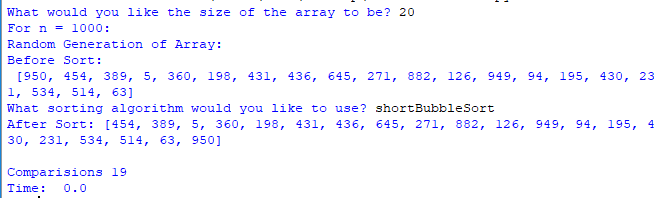
if (left <= n) and (A[left] > A[i]) O( log n)

max = left

else

max = i

if (right<=n) and (A[right] > A[max])

 max = right

if (max != i)

swap(A[i], A[max]) O (n ) comes from merging sorted lists

Heapify(A, max)

O (n log n)

# Summary of Sort Algorithm Complexities

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Sorting Algorithm | No. of comparisons | No. of  Copies | Time complexity | Space complexity |
| Bubble | ~ O(n2) | n(n-1)/2 | BC: O(N)  WC: O(n2) | O (1) |
| Selection | *~* | *~ 2n* | *BC:* O(n2)  WC: O(n2) | *O (1)* |
| Insertion | *~* n2 */ 4* | *~*n2 */ 4* | *BC:O(N)*  *WC:* O(n2) | *O (1)* |
| Merge | *~n log2n* | *~ 2n log2n* | *BC: O (n log2n)*  WC:*O (n log2n)* | *O(n)* |
| Quick | *\*BC: ~n log2n*  *\*WC: ~* n2 */2* | *BC: ~2n/3 log2n*  *WC: 0* | *BC: O (n log2n)*  *WC:* O (n2)*)* | *\*BC: O ()*  *\*WC: O(n)* |
| Heap | O () | O () | BC: O ()  WC: O () | WC: O (n) |

# Algorithm Analysis by Experimental Studies

# Sequencing Array Sorting Algorithms Based on their Complexity

## Bubble Sort

The bubble sort algorithm has an average performance of O(n2), the performance of this algorithm is relative to its size, so It would rarely be used to sort large data sets. Bubble sort works well on small sets of data where the inefficiency doesn’t affect the overall performance as much. The most efficient use of bubble sort would be for a list that is mostly sorted as time would be relative to the number of unsorted elements. One unsorted element would give 2n time, two would take 3n time and so on. In testing bubble ranked 6th in both time and comparisons making it the worst sorting algorithm.

## Insertion Sort

The insertion sort algorithm can be compared to an improved version of selection sort. It is quite efficient on smaller set of data and mostly sorted sets of data, because of this it is often used in conjunction with more complex algorithms. The downfall of insertion is that computationally expensive due to its shifting of all following elements when it shifts any element. In testing, Insertion ranked 5th in both time and comparisons making it the 2nd worst sorting algorithm.

## Selection Sort

The selection sort algorithm has O(n2) complexity, which is like the bubble sort as its performance is relative to its size. Selection sort is notable for how simple it is, and under specific circumstances has better performance than algorithms with greater complexity. It generally performs worse than much alike insertion sort. In testing, Selection ranks 4th for time and 1st every time for comparisons.

## . Quick Sort

The quick sort algorithm users a partition and pivot to sort elements and this gives it great linear time efficiency. Quick sort can be quite a complex algorithm but in practice it is quite fast. With its mediocre space usage of O (log n) all round quick sort is one of the top algorithms, although poor pivot selection can alter the performance significantly. In testing, Quick ranks 1st in time and 4th in comparisons.

## Merge Sort

The merge sort algorithms worst run time is O (n log n), this means that it scales well when sorting large sets of data. Merge sorts efficiency stems from when it combines list as this is one if its main operations. The O (n log n) run time is enables it to scale well with large sets of data. In testing, Merge ranks 2nd for lower values in time and 3rd for higher values, for comparisons it ranks 3rd every time.

According to the data obtained above, the best algorithm to use for the sorting of data would be the quick sort algorithm. Although quicksort has a rather high comparison count, it does have the best run time compared to the other sorting algorithms. Compared to the other algorithms quicksort comes third for the number of comparisons only beaten by Merge (second) and Heap (first) sort. The time taken to sort through data from sizes 1000 to 2000 does go up exponentially, it is still faster than the other algorithms.

The best approach to have when organising data would be to combine two sorting algorithms to assure that the best aspects of both the algorithms. For example the use of quick to organise the chunks while the data is merged as per merge sort (Sorting Algorithms).

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| A/N | 200 T | 200 C | 400 T | 400 C | 800 T | 800 C | 1000 T | 1000 C | 2000 T | 2000 C |
| B | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 |
| I | 5 | 4 | 5 | 4 | 4 | 4 | 4 | 4 | 4 | 4 |
| S | 4 | 5 | 4 | 5 | 5 | 5 | 5 | 5 | 5 | 5 |
| H | 2 | 1 | 3 | 1 | 3 | 1 | 3 | 1 | 3 | 1 |
| Q | 1 | 3 | 1 | 3 | 1 | 3 | 1 | 3 | 1 | 3 |
| M | 3 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |

*A =algorithm, N = number, C = comparisons, T = Time, B = Bubble, I = Insertion, S = Selection, H =Heap, Q= quick, M = Merge,*

|  |  |  |
| --- | --- | --- |
| A | Time (Ms) | Comparisons (Number) |
| B | 6th every instance | 6th every instance |
| I | 5th every instance  Except 3th Worst at 800 | 4th every instance |
| S | 4th most instances  Except 5th at 800, | 5th every instance |
| H | 2nd or 3rd every instance  2nd at 200,2000, 3rd at 400,800,1000 | 1st Best every instance |
| Q | 1st every instance | 3rd every instance |
| M | 2nd or 3rd every instance  2nd at 400,800,1000 3rd at 200,2000 | 2nd every instance |

class UnitMark:

def \_\_init\_\_(self, marks, next = None):

self.length=len(marks)-1

self.rep=self.\_\_str(marks)

self.marks = marks

self.next = next

def \_\_str(self, marks):

terms = ["(ID: "+str(marks[0])+ \

", A1: "+str(marks[1])+ \

", A2: "+str(marks[2]) + \

", exam: "+str(marks[3])] O(1) for each. (ID,A1,A2,EXAM)

return str(terms)

def \_\_eq\_\_(self, that):

return self.marks[0] == that[0]

def highest\_result(self):

if self is not None:

curr = self

curr\_marks = self.marks

curr\_ID = curr\_marks[0]

highest\_mark = curr\_marks[1]+ curr\_marks[2]+curr\_marks[3] O(N) for getting highest mark

if curr.next is not None:

while curr.next is not None:

curr = curr.next

totalMark=curr.marks[1]+curr.marks[2]+curr.marks[3] O(N) for getting highest mark

if totalMark > highest\_mark:

curr\_ID = curr.marks[0] O(1)

highest\_mark= totalMark

print("----- Student with the highest mark is: ==> ID: "+ str(curr\_ID) +\

" (total mark: " + str(highest\_mark) +")") O(1) adding variables

print()

else: # print student Id and highest total mark

print("Student with the highest mark is: ==> ID: "+ curr\_ID +\

" (total mark: " + str(highest\_mark) +")") O(1) adding variables

print()

else:

print ("no student record found!")

def insert\_unit\_result(self, st\_unit\_marks): # st\_unit\_marks, as a node, stores [ID, mark1, mark2, mark3] + pointer

new\_node =[0]\*4

old\_node =[0]\*4

old\_node[0]=self.marks[0]

old\_node[1]=self.marks[1]

old\_node[2]=self.marks[2]

old\_node[3]=self.marks[3]

new\_node[0]=st\_unit\_marks.marks[0]

new\_node[1]=st\_unit\_marks.marks[1]

new\_node[2]=st\_unit\_marks.marks[2]

new\_node[3]=st\_unit\_marks.marks[3]

if self is None:

return st\_unit\_marks

else:

previous = None

curr = self

while curr is not None:

curr\_marks = curr.marks

curr\_node\_id = curr.marks[0]

#print("current id: "+ str(curr\_node\_id)+" new id: " + str(new\_node[0])) O(1) for adding elements

if curr\_node\_id < new\_node[0]:

previous = curr

curr = curr.next

#print("-------curr\_node\_id > new\_node[0] - continue...")

elif curr\_node\_id == new\_node[0]:

curr.marks = st\_unit\_marks

return self

else:

st\_unit\_marks.next =curr

if previous is None:

return st\_unit\_marks

else:

previous.next = st\_unit\_marks

return self

if curr is None:

previous.next = st\_unit\_marks

return self

def reverse(self):

node = self

tsil = None

while node is not None:

tsil = UnitMark(node.marks, tsil)

node = node.next

return tsil

def delete(self, ID):

node = self

while node is not None:

if node.next == (ID,):

node.next = node.next.next

return

else:

node = node.next

def print\_one\_unit\_result(u\_list):

if u\_list is None:

print("No student records")

return

else:

print("Current unit result:")

lyst = u\_list.marks

total = lyst[1]+lyst[2] +lyst[3] O(1) for adding elements

print(" Student\_ID.: " + str(lyst[0])+" A1: " + str(lyst[1])+" A2: "

+ str(lyst[2])+" Eaxm: " + str(lyst[3])+" ->total " + str(total)+"\n") O(1) for adding elements

""" -----print all student marks of the unit-------- """

def print\_unit\_result(u\_list):

if u\_list is None:

print("There are no student records")

return

else:

print("The current linked list contains:")

curr = u\_list

while curr is not None:

lyst = curr.marks

total = lyst[1]+lyst[2] +lyst[3]

print(" Student\_ID.: " + str(lyst[0])+" A1: " + str(lyst[1])+" A2: "

+ str(lyst[2])+" Eaxm: " + str(lyst[3])+" ->total " + str(total)) O(1) for adding elements

curr = curr.next

print()

def reversePrint(u\_list):

if u\_list is None:

print("No student records")

return

else:

print("Printing list in reverse as specified:")

curr = u\_list

while curr is not None:

temp = list(curr.marks)

total = temp[1] + temp[2] + temp[3] O(1) Adding elements

if total < 50:

grade = "N."

elif total < 60:

grade = "C."

elif total < 70:

grade = "Cr."

elif total < 80:

grade = "D."

else:

grade = "Hd."

temp.append(total)

temp.append(grade)

outstr = ", ".join([str(elem) for elem in temp])

print("(" + outstr + ")")

curr = curr.next

print()

def main(size = 7):

unit\_node1 = None

unit\_node2 = UnitMark([1189, 2, 30, 30], unit\_node1)

unit\_node3 = UnitMark([1145, 9, 16, 20], unit\_node2)

unit\_node4 = UnitMark([1122, 11, 19, 40], unit\_node3)

unit\_node5 = UnitMark([1116, 8, 16, 35], unit\_node4)

unit\_node6 = UnitMark([1114, 14, 21, 30], unit\_node5)

unit\_node7 = UnitMark([1112, 10, 6, 50], unit\_node6)

unit\_node8 = UnitMark([1111, 17, 22, 30], unit\_node7)

unit\_list\_head = unit\_node8

print\_unit\_result(unit\_list\_head)

unit\_list\_head.highest\_result()

lyst = [1325, 17, 20, 20]

print("Insert a new record: ID: "+ str(lyst[0]) +" A1: "+ str(lyst[1])+ " A2: "+ str(lyst[2]) +" Exam: "+ str(lyst[3])+"\n") O(1) Adding elements

new\_unit\_node = UnitMark(lyst, None)

new\_unit\_list\_head = unit\_list\_head.insert\_unit\_result(new\_unit\_node)

print\_unit\_result(new\_unit\_list\_head)

print("The Reverse order is...")

reversePrint(new\_unit\_list\_head.reverse()) O(N) for reversing however many elements in array

val=input(int())

reversePrint(new\_unit\_list.delete())

if \_\_name\_\_ == "\_\_main\_\_":

main()

O(N^2) for O-Notation of code)

"Sorting Algorithms. https://resources.saylor.org/wwwresources/archived/site/wp-content/uploads/2011/06/Sorting-Algorithm.pdf"

Nazeer, K. A. and M. Sebastian (2009). Improving the Accuracy and Efficiency of the k-means Clustering Algorithm. Proceedings of the world congress on engineering, Association of Engineers London.

<https://resources.saylor.org/wwwresources/archived/site/wp-content/uploads/2011/06/Sorting-Algorithm.pdf>

[**https://www.quora.com/Which-sorting-algorithm-is-best-and-why**](https://www.quora.com/Which-sorting-algorithm-is-best-and-why)