CSP2348 Data Structures

Assignment 2: Algorithm Implementation and Analysis

- Topic: Algorithm Design, Analysis & Implementation

Objectives from the Unit Outline

- Apply mathematical theory to algorithm analysis.
- Design algorithms using various data structures.
- Analyse complexity and performance of their associated algorisms.

General Information:

- This is *an individual* assignment. Required skills/tasks include algorithm design, complexity analysis (to part of the questions) and implementation using Python programming language. (*Note*: If you want to use a programming language other than Python, please let your tutor know beforehand, and you may be required to demonstrate your work in the end. You must use one single programming language for all your programming tasks).

- Algorithm complexity analysis is one of the most important skills covered in this unit. It can be done by *theoretical algorithm analysis* or by *experimental study*. Generally, theoretical analysis is performed by applying the asymptotic theory/methods (such as in O-notation, etc.) to the algorithm's complexity function that reflects the number of basic operations the algorithm must execute on the input size, so the growth rate of the function can be estimated. On the other hand, the experimental study requires the implementation of the algorithms to reveal the growth trend of the complexity function.

- An array is a random-access structure, a typical data structure often used in many applications. You can use unique indexes to access array components in constant time. In term of data manipulation, this property makes array the most effective data structure.

- This assignment focuses on the design, analysis, implementation, and performance comparison of array-based sorting algorithms. It consists of three questions. The first question requires practicing algorithm design, algorithm analysis using O-notation, and algorithm implementation in Python. Question 2 is on the topic of *theoretical algorithm analysis*. You are required to give a summary of algorithm complexities and related features of the array-based sorting algorithms that are introduced/covered in this unit. Question 3 requires you to complete an *experimental study* for several array-based sorting algorithms.

- You will need to submit one assignment file (or report, in Word or PDF format) plus several Python code files.

Due: (See *Due* date/time in Assignments section on Canvas)

Marks: 100 marks (which will be converted to 30% of unit mark)

Tasks

Q1: Array sorting algorithm: design, implementation and analysis (35 marks)

(1) Bubble sort algorithm

Like *Insertion-sort* and *Selection-sort* algorithms, the *Bubble-sort* is an elementary array-sorting algorithm. The algorithm's name is a metaphor for the way air bubbles move through a bucket of water, with larger bubbles "bubbling" to the top of the water's surface faster than smaller ones.

The bubble sorting strategy can be described as follows:

> For a given array A[ ] of *n* (comparable) elements, the *bubble sort* algorithm scans the array n-1 times, where, in each scan, it keeps moving larger elements towards their final positions in (the sorted version of) the array. This guarantees that, in the end of the i-th scan, the i-th largest element is placed in A[n-i], *i* = 1, 2, …, n-1.
>
> Specifically, during the i-th scan, the algorithm performs n-1 *pair-wise* comparisons: for each array index k (k=0, 1, 2, …, n-2), it compares the current array element A[k] with the element after it, A[k+1], and swaps them if they are not in the required order (e.g., *ascending order*).

Based on the above sorting strategy,

(a) Write an algorithm (in pseudo code) for the bubble sort algorithm, which takes an array as input.

(b) Analyse your algorithm using O-notation by two ways, i.e.,
- by counting the number of comparisons; and
- by counting the number copying operations.

(c) Convert your algorithm to a Python code and test the code using an array of some 20 elements.

(2) Improvement to the bubble sort algorithm

Observation 1: In the bubble sort algorithm, after scanning the array for *k* times (*k* = 1, 2, …), the *k* largest elements are placed in their correct/final positions (of the sorted version of the array). So, when scanning the array for the *(k*+1)-th time, it is sufficient to scan only the subarray of the first *n-k* elements (i.e., there is no need to scan the entire array). In this way, the total number of comparisons can be reduced.

Based on this observation, modify the algorithm you did in step (1) (a), and then re-do (1) (b)~(c) accordingly.
(Note: to distinguish, the modified algorithm can be called *Obs1-BubbleSort* algorithm)

Observation 2: You may find that, if there is no "swapping" at all in the i-th scan to the array, the array is "already" sorted. In this case, no more scans are needed, and the algorithm can end. In this way, the total number of comparisons can be further reduced (for example, this version of the algorithm can be more efficient when the input array is "already" sorted or "nearly" sorted).

Based on this observation, modify the algorithm you completed in step (1) (a), and then re-do (1) (b)~(c) accordingly.

(*Hint*: set a flag (at the start of each scan) to monitor the swapping during the scan. If no swapping occurs during a scan, end the algorithm)
(Note: to distinguish, the modified algorithm can be called *Obs2-BubbleSort* algorithm)

Observation 3: By combining the ideas presented in the Observation 1 and 2, you can get a better version of the bubble sort algorithm because it may reduce a greater number of comparisons in general case.
Complete a final version of the bubble sort algorithm, which combines the work you've done in the steps listed in Observation 1 and 2.
(Note: to distinguish, you may call the modified version of the algorithm an *Obs3_BubbleSort* algorithm)

(3) Variation of the bubble algorithm

The bubble sorting algorithm can be modified in a few ways to improve the algorithm performance in some special situation/s. Consider the following two approaches:

(i). ***Sink-down* sort algorithm**

The sink-down sort algorithm is a variation of the bubble sort algorithm - it is named for the way smaller elements "sink" to the bottom of the list (instead of the way larger bubbles "bubbling" up to the water's surface). That is, in the i-th scan, instead of moving larger elements towards their final positions in (the sorted version of) the array, sink-down algorithm gradually moves *smaller elements* towards their final positions in (the sorted version of) the array and, in the end of the i-th scan, the algorithm places the i-th smallest element into A[i-1], i=1, 2, …n-1.

Based on this idea, modify your bubble sort algorithm completed in step (1) to complete the *sink-down sort* algorithm. That is, re-do step (1) (a) ~ (c) for the sink-down algorithm (i.e., re-do steps (1) (a) ~ (c) but replace "bubble" with "sink" and "bubbling" with "sinking", etc.).
Once this is done, improve your algorithm by applying the strategy you completed in the *Obs3_BubbleSort* algorithm.
(*Hint*: Unlike the bubble sort algorithm, in this algorithm you should do a cascade scan of the array from the last element to the first, not from the first to the last).

(ii). ***Bi-directional-bubble* sort (BDBS) algorithm**

The BDBS algorithm is another variation of the bubble sort algorithm. It combines the ideas of bubble sort and sink-down sort algorithm together and, instead of scanning the array from one-side to the other, it scans the array in alternate directions. Strategically, it scans the array from first to last in one scan and then scan it from the last to first in the next scan …. Or, the inner loop of the algorithm contains two scans to the array, one is the *left-to-right* scanning and the other is the *right-to-left* scanning. To be more efficient, in i-th scan, i = 1, 2, …, [n/2], the algorithm scans the subarray A[i-1, i, i+1, …, n-i] only:

- *left-to-right* scanning: the algorithm scans the array from first element to the last (or left to right) of the subarray A[i-1, i, i+1, …, n-i], compares pairs of adjacent elements and swaps if necessary (this implies that the largest element of the subarray A[i-1 … n-i] is placed into A[n-i]); and
- *right-to-left* scanning: the algorithm then scans the array from the last element to the first element of the subarray A[i-1, i, i+1, …, n-i-1], compares pairs of adjacent elements and swaps if necessary (implying that the smallest element of the subarray A[i-1 … n-i-1] is placed into A[i-1]).

The algorithm loops this way for about [n/2] times to sort the array.

Based on the strategy described above, modify the bubble sort algorithm you completed in step (1) (or by modifying sink-down sort algorithm you completed in step (3) (i)) to complete the BDBS algorithm.

Q2: Summary of theoretical analysis of array-based sorting algorithms (15 marks)

Review all arrays-based sorting algorithms that you have learnt so far (including the algorithm/s you've done in Q1) and fill in the missed complexity related data in Table 1, which has been partially completed (note that the complexities shown in Table 1 is of *Average case* unless noted otherwise). While no detailed algorithm analysis is required for this question, you are requested to fill in the table as detailed as you can.

Table 1: array sorting algorithm complexity (by *theoretical analysis*)

Sorting algorithm	**Number of comparisons**	**Number of copies**	**Time complexity**	**Space complexity**
Selection	$\sim n^2/2$	$\sim 2n$	$O(n^2)$	
Insertion	$\sim n^2/4$	$\sim n^2/4$	$O(n^2)$	
Merge	$\sim n \log_2 n$	$\sim 2n \log_2 n$	$O(n \log_2 n)$	
Quick	$^*BC : \sim n \log_2 n$ $^*WC: \sim n^2/2$	$^*BC :\sim 2n/3 \log_2 n$ $^*WC: 0$	$^*BC : O(n \log_2 n)$ $^*WC: O(n^2)$	$^*BC: O(\log_2 n)$ $^*WC: O(n)$
Heap				
Bubble				
Obs1-Bubble				
Obs2-Bubble				
Obs3-Bubble				
Sink-down				
Bi-directional				

Notes: *: BC – best case; WC- worst case

Q3: Algorithm analysis by *experimental study* (35 marks)

Write Python codes to implement a system that can be used to aid experimental study on array-based sorting algorithms. There are two subtasks:

(a) Develop a system to test various cases of sorting algorithms. The system starts with a main *Menu* with three menu options:

```
1. Test an individual sorting algorithm
2. Test multiple sorting algorithms
3. Exit
```

This allows the user to choose to test individual or multiple sorting algorithms.

(i). Test an individual sorting algorithm
This option leads to a sub-menu, like:

1. Test selection sort algorithm
2. Test insertion sort algorithm
3. Test merge sort algorithm
4. Test quick sort algorithm
5. Test heap sort algorithm
6. Test bubble sort algorithm
7. Test Obs1-bubble sort algorithm
8. Test Obs1-bubble sort algorithm
9. Test Obs1-bubble sort algorithm
A. Test Sink-down sort algorithm
B. Test Bi-directional sort algorithm

from which the user can choose to test any one specific sorting algorithm. Then
(Note: some sort algorithms are the ones that you have completed in Q1).

- The system prompts the user to enter the size of the array, i.e., an integer number *n* (*n*>0), and then it randomly generates *n* integers and stores them in the array, say *A[ ]*.
- Then it runs the sorting algorithm (taking *A[ ]* and *n* as input parameters), and outputs the number of comparisons (see step (2) in the *Recommended procedure*)
- Once completed, it goes back to the main menu.

(ii). Test multiple sorting algorithms

- The system prompts the user to enter an integer number *n* (*n*>0) as the size of an array, and then randomly generates *n* integers, and stores them in an array, say *A[ ].*

  (Note: For performance comparison, you must use the **same *A[…]*** and ***n*** to test all sorting algorithms in the following steps, i.e., do not re-generate array data for different sorting algorithms, why?).

- Print a table heading, like:

Sorting algorithm name	Array size	Num. of Comparisons	Run time (in ms.)

- Loop to generate a table: For each of the 11 sort algorithms (see those listed in step (a) (i)), the system calls individual sort algorithms, taking *n* and A[ ] as input/test data, and outputs the sorting results (including the name of the sorting algorithm, the size of the test data, the number of comparisons and the running time). That is, it produces a line of result, e.g., for the *selection* sort algorithm, the output line may look like:

selection sort	1000	499500	1.32 (ms.)

– Once completed, it goes back to the main menu.

(b) Conduct an *experimental study*: by completing the following Table 2 and Table 3.

This can be done either by a *manual* or an *automatic* way:

- *Manually*: Run the above code/s (e.g., see (a) (ii) above), one at a time, with appropriate parameters (e.g., with different *n* values), collect data from the outputs, do necessary calculations (e.g., averaging over 10 runs for a specific n value, etc.) and fill in the tables; or

- *Automatically*: write a code (e.g., by modifying the above code/s) so that it produces the two tables as the output.
  (Note: Your output format can be slightly different, but it must contain the required information, see those in the tables below).

Table 2: Experimental study: Average number of comparisons for sorting arrays of n integers (over 10 runs).

Sorting Algorithm	*n=100*	*n=200*	*n=400*	*n=800*	*n=1000*	*n=2000*
Selection						
Insertion						
Merge						
Quick						
Heap						
Bubble						
Obs1-Bubble						
Obs2-Bubble						
Obs3-Bubble						
Sink-down						
Bi-directional						

(**Go to the next page**)

Table 3: Experimental study: Average running time (in ms) for sorting arrays of n integers (over 10 runs).

Sorting Algorithm	*n=100*	*n=200*	*n=400*	*n=800*	*n=1000*	*n=2000*
Selection						
Insertion						
Merge						
Quick						
Heap						
Bubble						
Obs1-Bubble						
Obs2-Bubble						
Obs3-Bubble						
Sink-down						
Bi-directional						

Recommended procedure (for Q3)

To assist you prepare and achieve the solution/s, the following steps are recommended:

(1) Write Python codes to implement the following array-based algorithms that can be used to sort an array of *n* integers:
   - Selection sort
   - Insertion sort
   - Merge sort
   - Quick sort
   - Heap
   - Bubble
   - Obs1-Bubble
   - Obs2-Bubble
   - Obs3-Bubble
   - Sink-down
   - Bi-directional

   (Note: those algorithms have been implemented and tested in the lab sessions or implemented in Q1 - you should use them as your start points).

(2) For each of the above codes, make necessary modification to it so that it not only sorts the array, but also counts the number of comparisons, and records the running time (say, in *ms*).

(3) Write Python code to implement (a) (i).

(4) Write Python code to implement (a) (ii).

(5) Complete (b).

Other requirements (on coding):

1) This assignment requires using Python programming language to implement the coding tasks. If you wish to use a programming language other than Python, you need to get a permission from your tutor beforehand, and you may be required to demonstrate your work in the end.
2) The only package/s you can use (, or import) are *time*, *random*, and *sys*. You should not use/import any other package (such as *tabulate*, *numpy*, etc.) in your code/s. If you "import" any non-standard package/s into your code that prevent your code from running in our lab environment when marking your assignment, you should bear the risk of mark deduction. Therefore, you are recommended to test your codes in the lab before submission.
3) You should use one single programming language for all your programming tasks.

Submission Instructions (3 marks)

- Submit your Assignment 2 via Canvas assessment submission facility.
- Your submission should include the assignment main document (i.e., a report) and Python source codes. The main assignment document must be in report style, in Word or PDF format (see below for detailed format requirement). Pages must be numbered. Your source code file/s must be runnable in our lab environment.
- Your submission should be in a single compressed file (e.g., in .zip), which contains your main document/report and Python source code file/s, and any other support documents, if any. Rename the .zip file in a format of

  <student ID>_< last name>_<first name>_CSP2348_A2.zip.

  As an example, if your student ID is 12345678 and your name is Ben SMITH, the submission file should be named 12345678_ SMITH_ Ben_CSP2348_A2.zip

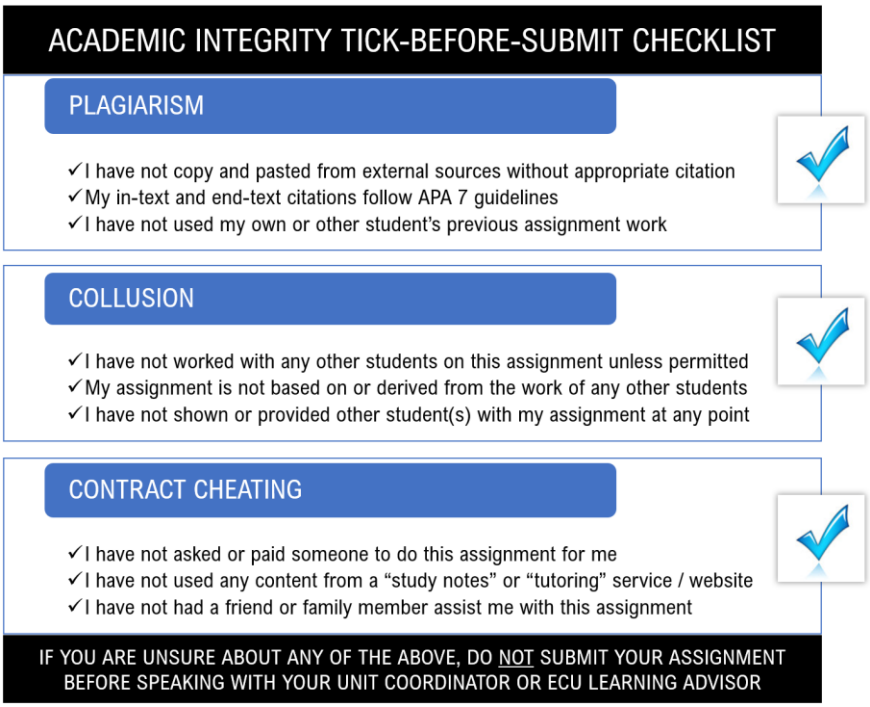
- No hard copy or email submission is acceptable.
- ECU rules/policies will apply for late submission of this assignment.

Format requirement of the Assignment report (12 marks):

Required Content	**Cover page** Must show unit code/name, assignment title, the student ID and name, due date etc.
	Executive Summary (*optional*) This should represent a snapshot of the entire report that your tutor will browse through and should contain the most vital information needed.
	Table of Contents (ToC) This must accurately reflect the content of your report and should be generated automatically in Microsoft Word with clear page numbers.
	Introduction Introduce the report, define its scope, and state any assumption if any ; Use in-text citation/reference where appropriate. **The main body** (note: you should use appropriate section title/s) This part should contain (but not limited to): • Understanding of concepts/techniques involved in the document/report. • Any strategies used to solve the problems (e.g., an approach to develop a solution, if any). • The questions being solved/answered, e.g., Q1: (should include but not limited to) (i) algorithms (in pseudo codes) and algorithm analysis; and (ii) screen shots (of running results of the code/s, e.g., one screenshot per algorithm). Q2: Table 1 completed. Q3: Complete Table 2 and 3; screen snapshots of execution results of code/s (e.g., one snapshot per sort algorithm showing the code's running result/s, including the number of comparison and algorithm running time), etc. • Comment/discussion or a critique of the solution developed/achieved if any. • No Python codes should be included/attached in the body of the report (all codes should be saved in separate files). **Conclusion** Outcomes or summary of the works done in this assignment. **References** A list of end-text reference, if any, formatted according to the ECU requirements using the APA format (Web references are also OK).
Other requirements	The document/report should be no more than 8 pages (excluding references, snapshots, and diagrams). The text must use font **Times New Roman** and **be not smaller than 12pt**.

Academic Integrity and Misconduct

- This assignment is an **individual** work (not a teamwork). Your entire assignment must be your own work (unless quoted otherwise) and produced for the current instance of the unit. Any use of uncited content that was not created by you yourself constitutes *plagiarism* and is regarded as Academic Misconduct. All assignments will be submitted to plagiarism checking software, which compares your submission version with previous copies of the assignment, and the work submitted by all other students in the unit (in the current semester or previous years/semesters).
- *Never give any part of your assignment to someone else*, even after the due date or the results are released. Do not work on individual assignment with other students. – You can help someone by explaining concepts, demonstrating skills, or directing them to the relevant resources. But doing any part of the assignment for them or with them or showing them your work is inappropriate. An unacceptable level of cooperation between students on an assignment is *collusion* and is deemed an act of Academic Misconduct. If you are not sure about plagiarism, collusion or referencing, simply contact your tutor or unit coordinator.
- You may be asked to explain and demonstrate your understanding of the work you have submitted. Your submission should accurately reflect your understanding and ability to apply the unit content and the skills learnt from this unit.
- Before submitting your assignment 2, make sure you have checked all items in the **Academic Integrity tick-before-submit checklist** below and watch the video by clicking the link next to the checklist image:



Watch this video before submitting your assignment

Indicative Marking Guide:

	Description	Allocated Marks (%)	Marks achieved & Comments
Q1	Bubble sort algorithm: – Design, analyse (of two ways), the code/s to implement the algorithm, test/running result (by snapshot/s) All code runs as expected? Outputs? & others	**15**	
	Improvement & variations (of bubble sort algorithm): – 3 improvement versions of the bubble sort algorithm: the algorithms & analysis; the code/s; test/running result – 2 variations of the bubble sort algorithm: the algorithms & analysis; the code/s; test/running result	**20**	
Q2	Algorithm analysis by theoretical studies - Table 1 completed?	**15**	
Q3	Algorithm analysis by experimental studies: - Implementation of sorting algorithms - 2-level menus implemented, - experimental data generation, correctly use of experimental data. - 2 tables completed. Code/s run as expected? Outputs correctly?	**35**	
Report	Document presented as per format requirement?	**12**	
	Submitted as per submission requirement? etc.	**3**	
Penalty	(*Academic misconduct? Late submission?*)		
Total mark of 100 (which will be converted to 30% of unit mark)		**100 (/30)**	