

School of Computing, Engineering and Mathematics (CEM)

Faculty of Engineering, Environment and Computing (EEC)

**5003CEM ADVANCED ALGORITHMS** | 2020

**PORTFOLIO OF CODE | REPORT**

**NAME: Guilherme Luis de Oliveira Miguel**

**SID: 8655712**

// After completion, **delete** comments like this, which appear in red, with ‘//’ in front

// **IMPORTANT!!** When you have finished, save your document with the following name. It does not matter if it is word or pdf – either is fine.   
 5003CEM\_REPORT\_YourFirstName\_YourSurname\_YourSID

// Obviously, replace YourFirstName, YourSurname, YourSID with the relevant details

// **PLEASE DO NOT UPLOAD A REPORT WHERE WE CANNOT IDENTIFY WHO YOU ARE!!**

// **For your coursework, you must complete 5 standard tasks, and 3 advanced tasks. But you do not need to write all of them up.**

**Choose any THREE standard tasks, and any TWO advanced tasks to write up.**

// When you have decided which tasks to write up, follow the structure and guidance below.

**1 STANDARD TASKS**

// Divide this section according to what tasks you have chosen to write up, for example:

* 1. **Standard Task 1: Selection Sort**
  2. **Standard Task 2: BST Find Methods**
  3. **Standard Task 3: Adjacency Matrix**

// Under each section:

1. Paste the commented, properly formatted code;
2. Building on the comments, explain how the code works;
3. Comment critically on your implementation. Consider elegance (shortness, compactness, conciseness), readability, and alternative implementations. Justify your choice of solution.

// What follows, in the box, is a **worked example**, using one of the non-assessed tasks. (NB the box is not necessary for your report; it’s just to show it’s an example).

**Standard Task 1: Selection Sort**

1. **Commented Code**

/\*

    SELECTION SORT

    input: unsorted array

    output: sorted array

    Implements selection sort with integral swap \*/

#include <iostream>

int \*swap(int first, int sub, int array[])

//This function swaps two members from an array

{

    int swap1 = array[first];

    int swap2 = array[sub];

    array[first] = swap2;

    array[sub] = swap1;

    return array;

}

int \*selectionSort(int arg[], int length)

{

    //This will loop through the array

    for(int i = 0; i < length; i++)

    {

        int indexMin = i;                                           //  Use first element as minimum

        int j = i + 1;                                             //  j as next element

        /\*

            When we get the value that we are testing as the min value we need to check if it is smaller than

the elements in front of it, if a smaller value is found then it needs to be put before the value tested.

        \*/

        for(j; j < length; j++)

        {

            if(arg[j] < arg[indexMin])                           // If j is less than the minimum

            {

                indexMin = j;                                     // j is now the minimum value

            }

        }

        /\*

            After checking the array the smaller value needs to be swaped with the value tested

        if the minimum value is i itself it will just stay the same

        \*/

        swap(i, indexMin, arg);                                  // Call swap funtion to rearrange array

    }

    return arg;

}

int main()

{

    int array[6] = {11, 22, 14, 67, 2, 9};                      // Our array to sort

    int length = (int)std::distance(std::begin(array), std::end(array));               // The lenght of the array is the distance between the first and last element

    /\*

        We need to get the size of the array in order to loop through it, it is a bit tricky to do it

    with c style arrays that's why I used std::distance, if a vector was used this would be able to be done

    in an easier way since vectors have a size property

    \*/

    int \*a = selectionSort(array, length);                     // Call the function to sort onto a pointer to an integer

    /\*

        In order to print the array after sorting we need a pointer that can point to each element.

        A way to surpass this would also be the use of a vector instead of a C style array.

    \*/

    std::cout << "Sorted array: ";

    for(int i = 0; i < length; i++)

    {

        if(i == length - 1)                                   // Checking where the full stop will be

            std::cout << a[i] << ". ";

        else

            std::cout << a[i] << ", ";                 // If i is different then the last element use comma

    }

    std::cout << std::endl;

    return 0;

}

A screenshot of a cell phone

Description automatically generated

1. **Explanation of Code**
2. **Critical Comment**

**Task 2**

1. **Commented Code**

/\*

    SELECTION SORT

    input: unsorted array

    output: sorted array

    Implements selection sort with integral swap \*/

#include <iostream>

int \*swap(int first, int sub, int array[])

//This function swaps two members from an array

{

    int swap1 = array[first];

    int swap2 = array[sub];

    array[first] = swap2;

    array[sub] = swap1;

    return array;

}

int \*selectionSort(int arg[], int length)

{

    //This will loop through the array

    for(int i = 0; i < length; i++)

    {

        int indexMin = i;                                           //  Use first element as minimum

        int j = i + 1;                                             //  j as next element

        /\*

            When we get the value that we are testing as the min value we need to check if it is smaller than

the elements in front of it, if a smaller value is found then it needs to be put before the value tested.

        \*/

        for(j; j < length; j++)

        {

            if(arg[j] < arg[indexMin])                           // If j is less than the minimum

            {

                indexMin = j;                                     // j is now the minimum value

            }

        }

        /\*

            After checking the array the smaller value needs to be swaped with the value tested

        if the minimum value is i itself it will just stay the same

        \*/

        swap(i, indexMin, arg);                                  // Call swap funtion to rearrange array

    }

    return arg;

}

int main()

{

    int array[6] = {11, 22, 14, 67, 2, 9};                      // Our array to sort

    int length = (int)std::distance(std::begin(array), std::end(array));               // The lenght of the array is the distance between the first and last element

    /\*

        We need to get the size of the array in order to loop through it, it is a bit tricky to do it

    with c style arrays that's why I used std::distance, if a vector was used this would be able to be done

    in an easier way since vectors have a size property

    \*/

    int \*a = selectionSort(array, length);                     // Call the function to sort onto a pointer to an integer

    /\*

        In order to print the array after sorting we need a pointer that can point to each element.

        A way to surpass this would also be the use of a vector instead of a C style array.

    \*/

    std::cout << "Sorted array: ";

    for(int i = 0; i < length; i++)

    {

        if(i == length - 1)                                   // Checking where the full stop will be

            std::cout << a[i] << ". ";

        else

            std::cout << a[i] << ", ";                 // If i is different then the last element use comma

    }

    std::cout << std::endl;

    return 0;

}

1. **Explanation of Code**
2. **Critical Comment**

**Task 5**

1. **Commented Code**

/\*

    SELECTION SORT

    input: unsorted array

    output: sorted array

    Implements selection sort with integral swap \*/

#include <iostream>

int \*swap(int first, int sub, int array[])

//This function swaps two members from an array

{

    int swap1 = array[first];

    int swap2 = array[sub];

    array[first] = swap2;

    array[sub] = swap1;

    return array;

}

int \*selectionSort(int arg[], int length)

{

    //This will loop through the array

    for(int i = 0; i < length; i++)

    {

        int indexMin = i;                                           //  Use first element as minimum

        int j = i + 1;                                             //  j as next element

        /\*

            When we get the value that we are testing as the min value we need to check if it is smaller than

the elements in front of it, if a smaller value is found then it needs to be put before the value tested.

        \*/

        for(j; j < length; j++)

        {

            if(arg[j] < arg[indexMin])                           // If j is less than the minimum

            {

                indexMin = j;                                     // j is now the minimum value

            }

        }

        /\*

            After checking the array the smaller value needs to be swaped with the value tested

        if the minimum value is i itself it will just stay the same

        \*/

        swap(i, indexMin, arg);                                  // Call swap funtion to rearrange array

    }

    return arg;

}

int main()

{

    int array[6] = {11, 22, 14, 67, 2, 9};                      // Our array to sort

    int length = (int)std::distance(std::begin(array), std::end(array));               // The lenght of the array is the distance between the first and last element

    /\*

        We need to get the size of the array in order to loop through it, it is a bit tricky to do it

    with c style arrays that's why I used std::distance, if a vector was used this would be able to be done

    in an easier way since vectors have a size property

    \*/

    int \*a = selectionSort(array, length);                     // Call the function to sort onto a pointer to an integer

    /\*

        In order to print the array after sorting we need a pointer that can point to each element.

        A way to surpass this would also be the use of a vector instead of a C style array.

    \*/

    std::cout << "Sorted array: ";

    for(int i = 0; i < length; i++)

    {

        if(i == length - 1)                                   // Checking where the full stop will be

            std::cout << a[i] << ". ";

        else

            std::cout << a[i] << ", ";                 // If i is different then the last element use comma

    }

    std::cout << std::endl;

    return 0;

}

1. **Explanation of Code**
2. **Critical Comment**

|  |
| --- |
| **GUIDANCE:**  // the code, comments and output do not count towards the word limit. This means that the answer below is 527 words. Your word limit is 3000 words, and there are 5 tasks to cover. So assuming 500 words each for Standard, you’d have 750 each for Advanced (altho’ these may require more). This would still be a string answer even without the material about C++.  1.1 **Bubble Sort**   1. **Commented Code**   '''BUBBLE SORT  input: unsorted list  output: sorted list  Implements bubble sort with integral swap  '''  def bubblesort(a):  swap = True #declare flag ‘swap’, set to true  while swap: #while the flag swap is true (outer loop)  swap = False #set the flag swap to false  for i in range (len(a)-1): #(inner loop)iterate through the unsorted list from start to #penultimate element  if a[i] > a[i+1]: #if 1st element value is greater than the element to the right (2nd)  swap = True #set swap flag to true, and swap the two elements  savedval = a[i+1] #create a buffer to save the 2nd element value, savedval  a[i+1] = a[i] #set 2nd element to be the value of the 1st  a[i] = savedval #set the 1st element to be the value of the 2nd  return a #when the while loop completes, return the sorted list  print(bubblesort([4,1,6,8,2,9,0]))    **GUIDANCE:**  // Use courier. The above is 10pt, but change the size until things look right. Courier is a non-proportional (or ‘monospaced’) font (i.e. all the characters take up the same horizontal width), which will preserve the tabbing / spacing correctly. Proportional fonts will not work.  // Make sure your code is **fully commented**. This has two purposes: (1) to show us you understand it; (2) so that others can understand it (including yourself)  // The docstring should appear in courier, in green.  // The comments should appear in courier, in red.  // Make sure your code is properly indented and that the comments do not wrap. Comments should be lined up. If comment is too long, leave a space in the code and tab the remainder of the comment over, as with the inner loop comment above. Where possible, avoid long comments. Note that the report is landscape format to allow space for code and comment formatting.  // Run your code, and paste the screenshotted output, so it appears after the code.   1. **Explanation of code**   This code implements bubble sort.  Bubble sort works by comparing each consecutive element with its rightward neighbour, and swapping them. This is done until the end of the list is reached. This what the inner loop in the code achieves.  When this process is finished, we start again at the beginning of the list and repeat. This is achieved through the outer loop above. The process is complete when there is nothing else to swap. If this is the case, the swap flag remains false, the while loop condition is not met, the code terminates, and the sorted list is returned.   1. **Critical comment**   This code is a straightforward implementation of bubble sort. However there are three things to consider. The first is, the code implements swap by using a buffer, savedval. In fact, Python does not require this. The swap can be achieved simply by doing:  a[i], a[i+1] = a[i+1], a[i]  However, this operation, while it can be done in Python, may not apply if other languages are used. For this reason, the standard logic for a swap has been used (in fact, Python also implements a buffer ‘behind the scenes’). In addition, the standard swap may be easier to understand, improving readability for someone not familiar with Python shortcuts.  The second issue is that this implementation is not as efficient as it could be, as it iterates through the entire list on each pass. While this is consistent with some descriptions of the algorithm, it is not necessary, as the sorted rightward elements increase with each pass, so Pass 1 means we have 1 sorted rightward element, Pass 2 means we have 2, and so on. In that case, if we have a list of 6 elements, and we have done 2 passes, we only need to consider 4 elements.  Therefore the code below implements a right-side lock, which, with each pass, shortens the iteration by 1. The result is that for a list of e.g. 6, instead of doing 25 comparisons, we only do 15, which is more efficient:  def bubblesort\_with\_lock(a):  swap = True  limit = 0  while swap:  limit = limit + 1  swap = False  for i in range (len(a)-limit):  if a[i] > a[i+1]:  swap = True  savedval = a[i+1]  a[i+1] = a[i]  a[i] = savedval  return a  print(bubblesort\_with\_lock([4,1,6,8,2,9]))  The third issue to be considered is language. My solution is in Python, and one reason for choosing this is it is faster to work with and results in less code, because the language is relatively high level. An alternative implementation, shown below in C++, is much longer. It requires a full swap implementation (including the buffer), as well as driver code (main). C++ is more strongly typed than Python, meaning that return types and data types must be specified. Equally, pointers need to be used as appropriate which increases the amount of work. Finally, using an array to represent the input sequence means we cannot just print the returned list, but need a print routine. This limitation means we might want to consider other data structures e.g. a vector.  In short, my solution is a simple, readable one with short code which was also fast to write. This does not necessarily make it the best, as the alternatives show, but ease of implementation, readability and speed are key issues for any coder and these are the criteria I have prioritised.  #include <iostream>  using namespace std;  void swap(int \*xp, int \*yp)  {  int temp = \*xp;  \*xp = \*yp;  \*yp = temp;  }  void bubbleSort(int arr[], int n)  {  int i, j;  for (i = 0; i < n-1; i++)  for (j = 0; j < n-i-1; j++)  if (arr[j] > arr[j+1])  swap(&arr[j], &arr[j+1]);  }    void printArray(int arr[], int size)  {  int i;  for (i = 0; i < size; i++)  cout << arr[i] << " ";  cout << endl;  }  int main()  {  int arr[] = {64, 34, 25, 12, 22, 11, 90, 0, 1}; //try different inputs  int n = sizeof(arr)/sizeof(arr[0]);  bubbleSort(arr, n);  printArray(arr, n);  return 0;  }  **GUIDANCE:**  // This is quite a long writeup for a standard task – and the reason for this is the critical comment. Remember the marking criteria (see Coursework assignment): 40% code (does it work, what’s its quality); 40% explanation (do you understand how it works); 10% critical comment; 10% clear communication. So in percentage terms it is best to prioritise the areas with bigger percentages. However, good critical comment can lift grades into First territory; they often go with good explanation. |

**2 ADVANCED TASKS**

// Format this section exactly as for Section 1, e.g.

* 1. **Advanced Task 1: BST remove method**
  2. **Advanced Task 2: <your choice>**

// Under each section, again, have three subsections:

1. Commented code
2. Code explanation
3. Critical comment

// THERE IS NO NEED FOR ANY REFERENCES SECTION OR CONCLUSION. THE REPORT IS FINISHED AT THIS POINT, BUT BE AWARE:

**IF YOU TAKE CODE FROM THE INTERNET / ANYWHERE ELSE YOU MUST ACKNOWLEDGE IT. GIVE THE URL / OTHER SOURCE AT THE TOP OF THE CODE IN THE DOCSTRING. IF YOU TAKE CODE WHOLESALE, VERBATIM, IT WILL BE WORTH NO CREDIT IF ATTRIBUTED. IF NOT ATTRIBUTED, AND THERE ARE ATTEMPTS TO MASK THIS WITH E.G. FUNCTION AND VARIABLE NAME CHANGES, THIS WILL BE AN ACO CASE FOR PLAGIARISM. YOU CAN LOSE ALL MARKS FOR THE MODULE, AND IF IT IS A REPEAT OFFENCE THE CONSEQUENCES CAN BE EVEN MORE SERIOUS.**

**Advanced Task 1: BST remove method**

**Advanced Task 3: concurrent news headline printer**