Question 1 1 .... 0 (3 marks)

Assume that the elements in an array of size 77 are already arranged in **descending** order (i.e. from largest value to smallest value). Answer the following:

1.1 How many data moves will insertion sort require to sort the array into **ascending** order (i.e. from smallest value to largest value)? (1)

Worst case $A^2 = (77)^2 = 5929$ $\frac{77 \times 76}{2} = 2926$

1.2 How many comparisons amongst the elements will selection sort perform in order to sort the array into **ascending** order (i.e. from smallest value to largest value)? (1)

Worst case $A^2 = (77)^2 =$ ~~59~~ 5929 2926

1.3 How many data moves will bubble sort require to sort the array into **ascending** order? (i.e. from smallest value to largest value) (1)

Worst case $A^2 = (77)^2 =$ ~~5929~~ 2926

Question 2 2 .... (4 marks)

Assume that the following array must be sorted:

```
[1020, 793, 340, 550, 811, 99, 100, 99, 12, 7, 60, 31, 340, 46]
```

3

Given that there exists a program where different methods implement different sorting algorithms and each method outputs the array after certain steps of the algorithms have been executed. Give the sorting algorithms which will generate each of the following output traces:

2.1 (1)

```
[1020, 793, 340, 550, 811, 99, 100, 99, 12, 7, 60, 31, 340, 46]
[793, 1020, 340, 550, 811, 99, 100, 99, 12, 7, 60, 31, 340, 46]
[793, 1020, 340, 550, 811, 99, 100, 99, 12, 7, 60, 31, 340, 46]
[340, 550, 793, 1020, 811, 99, 100, 99, 12, 7, 60, 31, 340, 46]
[340, 550, 793, 1020, 99, 811, 100, 99, 12, 7, 60, 31, 340, 46]
[340, 550, 793, 1020, 99, 100, 811, 99, 12, 7, 60, 31, 340, 46]
[99, 100, 340, 550, 793, 811, 1020, 99, 12, 7, 60, 31, 340, 46]
[99, 100, 340, 550, 793, 811, 1020, 12, 99, 7, 60, 31, 340, 46]
[99, 100, 340, 550, 793, 811, 1020, 12, 99, 7, 60, 31, 340, 46]
[99, 100, 340, 550, 793, 811, 1020, 7, 12, 60, 99, 31, 340, 46]
[99, 100, 340, 550, 793, 811, 1020, 7, 12, 60, 99, 31, 340, 46]
[99, 100, 340, 550, 793, 811, 1020, 7, 12, 60, 99, 31, 46, 340]
[99, 100, 340, 550, 793, 811, 1020, 7, 12, 31, 46, 60, 99, 340]
[7, 12, 31, 46, 60, 99, 99, 100, 340, 340, 550, 793, 811, 1020]
```

Merge Sort

(1)

2.2
```
[1020, 793, 340, 550, 811, 99, 100, 99, 12, 7, 60, 31, 340, 46]
[7, 1020, 793, 340, 550, 811, 99, 100, 99, 12, 31, 60, 46, 340]
[7, 12, 1020, 793, 340, 550, 811, 99, 100, 99, 31, 46, 60, 340]
[7, 12, 31, 1020, 793, 340, 550, 811, 99, 100, 99, 46, 60, 340]
[7, 12, 31, 46, 1020, 793, 340, 550, 811, 99, 100, 99, 60, 340]
[7, 12, 31, 46, 60, 1020, 793, 340, 550, 811, 99, 100, 99, 340]
[7, 12, 31, 46, 60, 99, 1020, 793, 340, 550, 811, 99, 100, 340]
[7, 12, 31, 46, 60, 99, 99, 1020, 793, 340, 550, 811, 100, 340]
[7, 12, 31, 46, 60, 99, 99, 100, 1020, 793, 340, 550, 811, 340]
[7, 12, 31, 46, 60, 99, 99, 100, 340, 1020, 793, 340, 550, 811]
[7, 12, 31, 46, 60, 99, 99, 100, 340, 340, 1020, 793, 550, 811]
[7, 12, 31, 46, 60, 99, 99, 100, 340, 340, 550, 1020, 793, 811]
[7, 12, 31, 46, 60, 99, 99, 100, 340, 340, 550, 793, 1020, 811]
[7, 12, 31, 46, 60, 99, 99, 100, 340, 340, 550, 793, 811, 1020]
```

Bubble sort ✓

(1)

2.3
```
[1020, 793, 340, 550, 811, 99, 100, 99, 12, 7, 60, 31, 340, 46]
[793, 1020, 340, 550, 811, 99, 100, 99, 12, 7, 60, 31, 340, 46]
[340, 793, 1020, 550, 811, 99, 100, 99, 12, 7, 60, 31, 340, 46]
[340, 550, 793, 1020, 811, 99, 100, 99, 12, 7, 60, 31, 340, 46]
[340, 550, 793, 811, 1020, 99, 100, 99, 12, 7, 60, 31, 340, 46]
[99, 340, 550, 793, 811, 1020, 100, 99, 12, 7, 60, 31, 340, 46]
[99, 100, 340, 550, 793, 811, 1020, 99, 12, 7, 60, 31, 340, 46]
[99, 99, 100, 340, 550, 793, 811, 1020, 12, 7, 60, 31, 340, 46]
[12, 99, 99, 100, 340, 550, 793, 811, 1020, 7, 60, 31, 340, 46]
[7, 12, 99, 99, 100, 340, 550, 793, 811, 1020, 60, 31, 340, 46]
[7, 12, 60, 99, 99, 100, 340, 550, 793, 811, 1020, 31, 340, 46]
[7, 12, 31, 60, 99, 99, 100, 340, 550, 793, 811, 1020, 340, 46]
[7, 12, 31, 60, 99, 99, 100, 340, 340, 550, 793, 811, 1020, 46]
[7, 12, 31, 46, 60, 99, 99, 100, 340, 340, 550, 793, 811, 1020]
```

Selection sort ✓

(1)

2.4
```
[1020, 793, 340, 550, 811, 99, 100, 99, 12, 7, 60, 31, 340, 46]
[7, 793, 340, 550, 811, 99, 100, 99, 12, 1020, 60, 31, 340, 46]
[7, 12, 340, 550, 811, 99, 100, 99, 793, 1020, 60, 31, 340, 46]
[7, 12, 31, 550, 811, 99, 100, 99, 793, 1020, 60, 340, 340, 46]
[7, 12, 31, 46, 811, 99, 100, 99, 793, 1020, 60, 340, 340, 550]
[7, 12, 31, 46, 60, 99, 100, 99, 793, 1020, 811, 340, 340, 550]
[7, 12, 31, 46, 60, 99, 100, 99, 793, 1020, 811, 340, 340, 550]
[7, 12, 31, 46, 60, 99, 99, 100, 793, 1020, 811, 340, 340, 550]
[7, 12, 31, 46, 60, 99, 99, 100, 793, 1020, 811, 340, 340, 550]
[7, 12, 31, 46, 60, 99, 99, 100, 340, 1020, 811, 793, 340, 550]
[7, 12, 31, 46, 60, 99, 99, 100, 340, 340, 811, 793, 1020, 550]
[7, 12, 31, 46, 60, 99, 99, 100, 340, 340, 550, 793, 1020, 811]
[7, 12, 31, 46, 60, 99, 99, 100, 340, 340, 550, 793, 1020, 811]
[7, 12, 31, 46, 60, 99, 99, 100, 340, 340, 550, 793, 811, 1020]
```

heap sort ✗

1020
/ \
793 340

Question 3 3. .... (3 marks) 2

Assume that the following unsorted array of integer values is given:

0	1	2	3	4	5	6	7	8	9	10	11	12	13
1020	793	340	550	811	99	100	99	12	7	60	31	340	46

Assume that the array needs to be sorted into ascending order (i.e. from smallest value to largest value) using **shell sort**. The specific sorting algorithm used for sorting sub-arrays is not important. Answer the following questions:

3.1 What would be the state of the given array after the first pass has been completed using a gap size of **7**? (1) ✓

99, 12, 7, 60, 31, 99, 46, 1020, 793, 340, 550, 811, 340, 100

3.2 What would be the state of the previous array (yielded by 3.1 above) after the second pass has been completed using a gap size of **4**? (1)

31, 12, 7, 60, 99, 99, 46, 811, 793, 340, 550, 1020, ~~340~~ 100

7 12 31 60 46 99 99 100 340 340 550 811 793 1020

3.3 What would be the state of the previous array (yielded by 3.2 above) after the third pass has been completed using a gap size of **2**? (1) ✓

7, 12, 31, 60, 46, 99, 99, 100, 340, 340, 550, ~~793~~, ~~811~~, 811, 793, 1020

Question 4 4. .... (4 marks) 1

Each of the following questions group various sorting algorithms together. For each question you have to identify a feature/characteristic shared by the algorithms that distinguishes them from other sorting algorithms.

4.1 Merge sort, radix sort and counting sort. (1)

~~Merge sort will~~ Radix sort will put values in ~~the~~ with the same digit ~~at the~~ position together
don't explain
Counting will put a correct value in its spot with each iteration
merge moves elements on one side(left)

4.2 Counting sort and quick sort. (1)

quick sort will put a value on its correct position with values on its left less than it and those on it's right have higher values
Don't explain sort!
Counting puts correct element one at a time with each iteration

(1)

4.3 Selection sort and heap sort.

selection sort swaps the smallest element to its correct position; without ~~heap sort~~ disturbing the array while heap sort takes the smallest while moving other elements in the array ✓

(1)

4.4 Comb sort, heap sort and quick sort.

Comb sort ~~has a const~~ swaps values separated by a gap

Don't explain sort!

heap sort heapifies ~~a~~ value to it's correct position, quick sort uses a value as a pivot and it'll be in its correct position ~~ma~~ maintaining a sorted property where elements on its left are less in value and vice ~~[illegible]~~ of the right

Question 5 (11 marks)

5.1 Imagine that a person's information is stored in a hash table. The hash function uses the concatenated ASCII codes of each person's surname to produce a numeric value, which is then scaled into the correct range using the modulus operator. Identify whether or not this hash function and hash key are a good choice, and justify your answer. Note that if you do not provide a justification, you will receive no marks for your answer. (2)

hash function is not good because multiple persons could share the same surname, the use use of ascii isn't good either because ~~mult~~ ~~differen~~ surnames aren't unique. The use of modulus is good because it keeps within a desired range from zero

5.2 Briefly explain why the size of a hash table is usually chosen to be a large prime number. (2)

Because prime numbers aren't divisible ✓ ergo provide a wider spread of hash keys because modulus won't produce a zero as much

1

5.3 Coalesced hashing combines probing with chaining. It is possible to include a cellar when using coalesced hashing, which is a reserved part of the table for the storage of colliding keys. An important decision is the size of the cellar. Answer the questions that follow:

a) Briefly describe what the result will be if the cellar is too small. (2)

Space would run out quickly and we would have to use probing all ✓ the time

1

5

b) Briefly describe what the result will be if the cellar is too large. (1)

Too large would be a waste of space ✓ if there isn't enough collisions and wastes space for efficient hashes

5.4 Given the following skeleton of a **HashTable** class, implement the **insert** method assuming that a division hash function with linear probing is used. Note that the size of the table is a prime number. Assume that the computeKey method returns a numeric key that represents the object obj. Also assume that the table size is fixed (i.e. no rehashing is required). (4)

```
public class HashTable {
        private Object table[];

        HashTable() {
                table = new Object[113];
        }

        private int computeKey(Object obj) {
                // return an appropriate hash for the provided key
        }

        public void insert(Object obj) {
                // your implementation goes here
        }
}
```

2

```
private
~~private int computekey (Object obj){
    return (int) obj;
}~~
public void insert(Object obj){ ✓
  int key = computekey(obj); key = key % 113
  if( table[key] == null){ table[key] = obj} ✓
  else { for(int i = 1; i < 113; i++ ){ ✓
            if( ~~table~~ table[~~[illegible]~~ key ~~%113~~ + i]==null)
              { table[key % 113 +i]= obj;}
     }
   }
}
```

hard coded

null

8

Question 6 6. . . . . (8 marks)

Assume that the following list of words must be hashed from top to bottom (**in the given order**, assuming the list is already frequency-sorted) with Cichelli's algorithm:

`Avaritia`	8
`Acedia`	6
`Invidia`	7
`Ira`	3
`Gula`	4

T = 9

Perform Cichelli's algorithm with `max = 3` on the list of words in the given order, and answer the following:

6.1 Fill in the hash values for each of the words in the table below: (5) 5

Word	Hash Value	
`Avaritia`	3	✓
`Acedia`	1	✓
`Invidia`	4	✓
`Ira`	0	✓
`Gula`	2	✓

6.2 Fill in the the g-values for each of the characters in the table below: (3) 3

Word	g-value	
A	0	✓
I	2	✓
G	3	✓

4

Question 7 7. . . . . (6 marks)

DNA strings are made up of four symbols: `A`, `C`, `G`, `T`. The probability of each symbol across multiple sequences is calculated, and is given below:

A	C	G	T
0.31	0.23	0.29	0.17

The DNA sequences are to be compressed with Huffman encoding. Answer the questions below:

7.1 Are the codes below a valid encoding for single-character DNA symbol compression? Motivate your answer. (2)

0

A	C	G	T
1	10	01	111

Yes these are valid because A having the highest probability has the lowest bit length and each code word corresponds to only one symbol

4

7.2 What is the **minimal average length** of a Huffman code that is necessary to store the single-character DNA symbols, according to the entropy of the data? Show all calculations. (1)

$$H(S) = -\Sigma p_i \log_2 p_i = -.31 \times \log_2(.31) - .23 \times \log_2(.23) - .29 \times \log_2(.29) - .17 \times \log_2(.17) = 1.96395$$ ✓

7.3 What is the **actual average codeword length** of the resulting Huffman coding? Show all calculations. (1)

$$H_{avg} = .31 \times 2 + .29 \times 2 + .23 \times 2 + .17 \times 2 = 2$$ ✓

7.4 What is the efficiency of the Huffman compression for the single-character DNA symbols? Show all calculations. (1)

$$\frac{H(S)}{H_{avg}} = \frac{1.96395}{2} = .98198 \text{ or } 98.1976\%$$ ✓

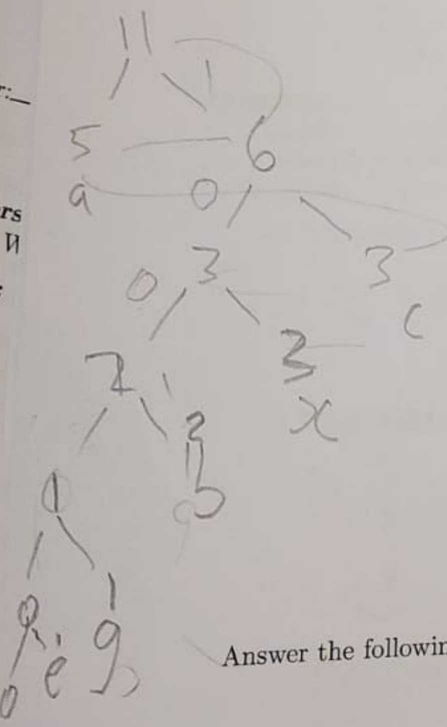
7.5 What could potentially improve the efficiency of the Huffman compression applied to DNA strings? (1)

We could try encoding pairs of ~~multip~~ ~~codons~~ characters ✓

2

Question 8 8 .... (6 marks)

Consider the following adaptive Huffman tree:



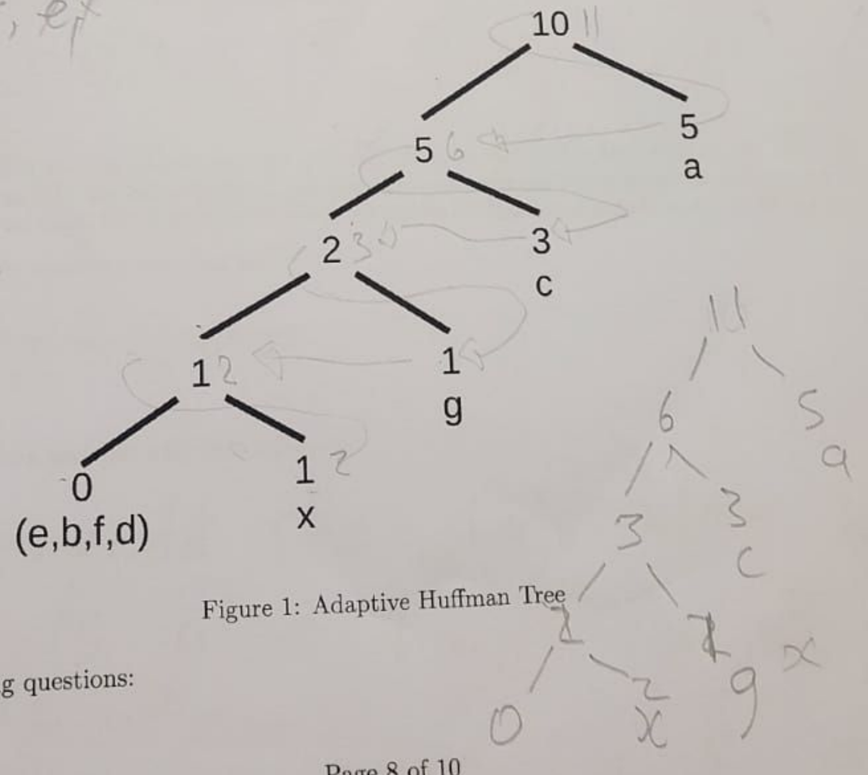


Figure 1: Adaptive Huffman Tree

Answer the following questions:

8.1 What is the code for the symbol **b**? (1)

0000 110 ✓

8.2 What is the code for the symbol **x**? (1)

0001 ✓

8.3 Assume the following symbols are all encoded using the adaptive Huffman tree in *Figure 1*, above, in the given order:

```
x, b, b, e, x
```

a) What is the code word for the symbol **b** now? (1)

1001 ✗

b) What is the code word for the symbol **x** now? (1)

101 ✗

c) What is the code word for the symbol **e** now? (1)

100001 ✗

d) What is the code word for the symbol **f** now? (1)

1000000110 ✗

Question 9 9. . . . . (2 marks) 1

Consider run-length encoding of the sequence of symbols ABCABCABC. According to the usual definition of run-length encoding, this sequence cannot be encoded because no runs of the same single character exist in the sequence. Assume we allow the elements of a run to be triples of characters and we use % as the escape character.

9.1 What would the resulting encoding be? 1 (1)

%3ABC ✓

9.2 What would be a problem with this approach? 0 (1)

This may add characters along the way and is also hard to read and encode why...

Question 10 10. . . . . 2 (3 marks)

A version of the **findNext** method for the *Knuth-Morris-Pratt* algorithm is given in pseudocode below:

```
findNext(pattern P, arr[] next)
{
        next[0] = -1;                         9
        i = 0;
        j = -1;               8
        while (i < P.length-1) {
                while (j == -1 || (i < P.length-1 && P[i]==P[j])) {
                        i++;
                        j++;
                        next[i] = j;
                }
                j = next[j];
        }
}
```

i = 0

i = 1

j = -1

[-1, 0,

Apply the above algorithm and give the array next for the following pattern: xyzxyzyxx

[-1, 0, 0, 0, 1, 2, 3, 1, 1]

✓ ✓ ✗

x y z x y z y x x

0 1 2 3 4 5 6 7 8