

Family Name:

Other Names:

ID Number:

Signature

COMP 103: Test 2

14 Sept, 2012

Instructions

- Time allowed: **35 minutes**
- There are 25 marks in total.
- Answer **all** the questions.
- The appendix contains the sorting algorithms and brief collection class documentation.
- If you think some question is unclear, ask for clarification.
- This test will contribute 10% of your final grade,
(But your mark will be boosted up to your exam mark if that is higher.)
- You may use paper translation dictionaries, and calculators without a full set of alphabet keys.
- Write your answers in the boxes in this test paper and hand in all sheets. You may ask for additional paper if you need it.
- You may write notes and working on this paper, but make sure it is clear where your answers are.

Questions

Marks

1. Sorting and Searching Algorithms

[10]

2. Implementing a Collection

[10]

3. Recursive Algorithms

[5]

TOTAL:

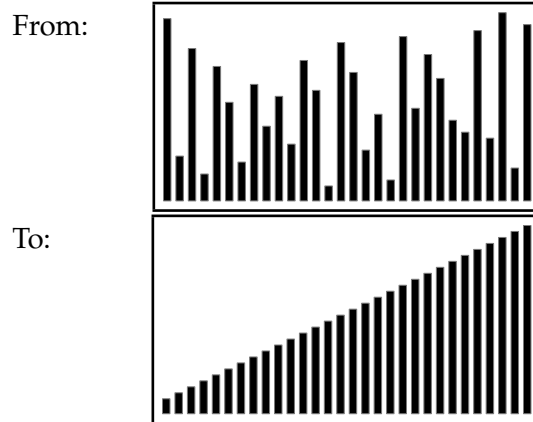
SPARE PAGE FOR EXTRA ANSWERS

Cross out rough working that you do not want marked.
Specify the question number for work that you do want marked.

Question 1. Sorting Algorithms

[10 marks]

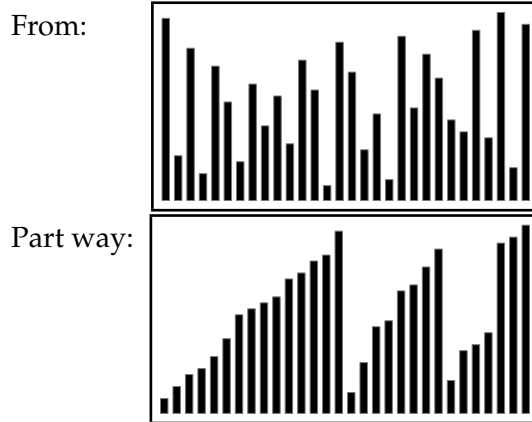
Suppose we want to sort a collection of sticks according to height:



Each of the following diagrams show the state of the sticks part way through sorting using a different sorting algorithm. For each diagram, say which algorithm is being used, and give a brief explanation for your choice.

The possible algorithms are selection sort, bubble sort, insertion sort, recursive merge sort, and quicksort.

(a) [2 marks]



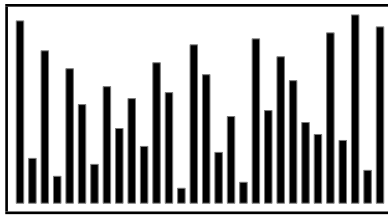
Sorting Algorithm:

Explanation:

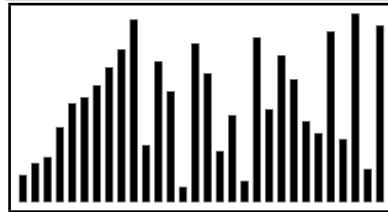
(Question 1 continued on next page)

(Question 1 continued)

(b) [2 marks] From:



Part way:

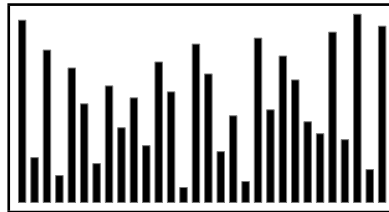


Sorting Algorithm:

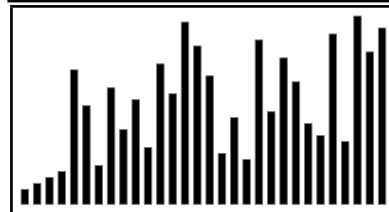
Explanation:

Empty box for the answer to part (b).

(c) [2 marks] From:



Part way:



Sorting Algorithm:

Explanation:

Empty box for the answer to part (c).

(Question 1 continued on next page)

(Question 1 continued)

(d) [2 marks] Under what circumstances will it be faster to use Insertion Sort to sort an array than to use QuickSort?

(e) [2 marks] Explain what it means for a sorting algorithm to be stable, and give one example of a stable sorting algorithm and one example of an unstable sorting algorithm.

Explanation:

An example stable algorithm:

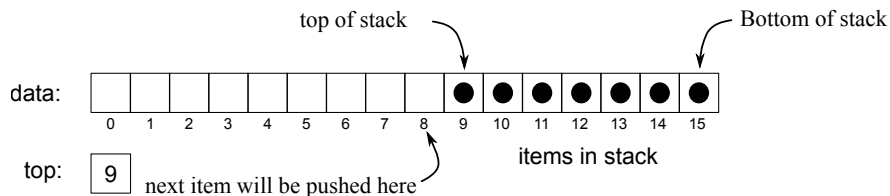
An example unstable algorithm:

Question 2. Implementing a Collection

[10 marks]

This question concerns a `YarraStack` class which implements a `Stack` collection using an array. `YarraStack` objects have a `data` field to hold the array of values, and a `top` field to hold the position of the value at the top of the stack. It stores the items at the right end of the array, (rather than from the beginning of the array), so the item at the bottom of the stack (the first item added) will be at position `length-1`.

The following diagram shows a stack with seven values in it.



Part of the `YarraStack` class is shown below:

```
public class YarraStack <E> extends AbstractCollection <E> {  
  
    private static final int INITIALCAPACITY = 8;  
    private E[] data;  
    private int top;  
  
    public YarraStack(){  
        data = (E[] ) new Object[INITIALCAPACITY];  
        top = data.length;           // Stack is initially empty  
    }  
  
    public E peek(){  
        if (top==data.length) throw new EmptyStackException();  
        return data[top];  
    }  
}
```

(a) [2 marks] Complete the following `size` method which should return the number of items in the stack. (Note, it must compute the size from the values in the fields.)

```
public int size () {  
  
  
  
  
  
  
  
  
  
}
```

(Question 2 continued on next page)

SPARE PAGE FOR EXTRA ANSWERS

Cross out rough working that you do not want marked.
Specify the question number for work that you do want marked.

Question 3.

[5 marks]

Consider the following `choose` method which returns a value from an array. It uses the `partition` method from QuickSort (included in the appendix).

```
public String choose(String[] data, int low, int high, int choice){
    if (low+1 == high) { return data[low]; }
    int mid = partition (data, low, high);
    if (choice < mid) { return choose(data, low, mid, choice); }
    else { return choose(data, mid, high, choice); }
}
```

(a) [1 mark] If the array `myData` contains the following strings, what value will be returned by `choose(myData, 0, 10, 7)`

Hint: show any working to get partial credit.

owl	rat	fox	vet	elk	hog	zoo	yak	jay	pig
0	1	2	3	4	5	6	7	8	9

(b) [2 marks] Explain the function of *choose* - what it does in general.

(c) [2 marks] State the average cost of `choose` when called on an array with n values and justify your answer.

You may express your answer using big(O) notation. State whether the cost depends on the value of the last argument (`choice`).

Appendix: Selection, Insertion, Bubble, QuickSort

```
public void selectionSort(String[] data){
    for (int place=0; place<data.length-1; place++){
        int minIndex = place;
        for (int sweep=place+1; sweep<data.length; sweep++){
            if (data[sweep].compareTo(data[minIndex]) < 0)
                minIndex=sweep;
        }
        swap(data, place, minIndex);
    }
}

public void insertionSort(String[] data){
    for (int i=1; i<data.length; i++){
        String item = data[i];
        int place = i;
        while (place > 0 && item.compareTo(data[place-1]) < 0){
            data[place] = data[place-1];    // move up
            place--;
        }
        data[place]= item;
    }
}

public void bubbleSort(String[] data){
    for (int top=data.length-1; top>0; top--){
        for (int sweep=0; sweep<top; sweep++){
            if (data[sweep].compareTo(data[sweep+1]) >0)
                swap(data, sweep, sweep+1);
        }
    }
}

public void quickSort(String[] data, int low, int high) {
    if (high-low < 2)    // only one item to sort.
        return;
    else {    // split into two parts, mid = index of boundary
        int mid = partition (data, low, high);
        quickSort(data, low, mid);
        quickSort(data, mid, high);
    }
}

private int partition (String[] data, int low, int high) {
    String pivot = data[(low+high)/2];
    int left = low-1;
    int right = high;
    while( left < right ) {
        do { left ++; } while ( left < high && data[left].compareTo(pivot) < 0);
        do { right --; } while (right >= low && data[right].compareTo(pivot) > 0);
        if ( left < right ) { swap(data, left , right ); }
    }
    return left ;
}
```

Appendix: MergeSort

```
public void mergeSort(String[] data) {
    String[] other = new String[data.length];
    for (int i=0; i<data.length; i++) { other[i]=data[i]; }
    mergeSort(data, other, 0, data.length);
}

public void mergeSort(String[] data, String[] temp, int low, int high) {
    if (low < high-1) {
        int mid = ( low + high ) / 2;
        mergeSort(temp, data, low, mid);
        mergeSort(temp, data, mid, high);
        merge(temp, data, low, mid, high);
    }
}

public void merge(String[] from, String[] to, int low, int mid, int high) {
    int index = low; //where we will put the item into "to"
    int indxLeft = low; //index into the lower half of the "from" range
    int indxRight = mid; // index into the upper half of the "from" range
    while (indxLeft<mid && indxRight < high) {
        if ( from[indxLeft].compareTo(from[indxRight]) <=0 )
            to[index++] = from[indxLeft++];
        else
            to[index++] = from[indxRight++];
    }
    // copy over the remainder. Note only one loop will do anything.
    while (indxLeft<mid)
        to[index++] = from[indxLeft++];
    while (indxRight<high)
        to[index++] = from[indxRight++];
}
```

Appendix: Collections

interface *Collection*<E>

```
public boolean isEmpty()
public int size()
public boolean add(E item)
public boolean contains(Object item)
public void remove(Object element)
public Iterator <E> iterator()
```

interface *List*<E> **extends** *Collection*<E>

```
// Implementations: ArrayList, LinkedList
public E get(int index)
public void set(int index, E element)
public void add(int index, E element)
public void remove(int index)
// plus methods inherited from Collection
```

interface *Set* **extends** *Collection*<E>

```
// Implementations: ArraySet, HashSet, TreeSet
// methods inherited from Collection
```

interface *Queue*<E> **extends** *Collection*<E>

```
// Implementations: ArrayQueue, LinkedList
public E peek () // returns null if queue is empty
public E poll () // returns null if queue is empty
public boolean offer (E element)
```

class *Stack*<E> **implements** *Collection*<E>

```
public E peek () // returns null if stack is empty
public E pop () // returns null if stack is empty
public E push (E element)
```

interface *Map*<K, V>

```
// Implementations: HashMap, TreeMap, ArrayMap
public V get(K key) // returns null if no such key
public V put(K key, V value) // returns old value, or null
public V remove(K key) // returns old value, or null
public boolean containsKey(K key)
public Set<K> keySet()
```

public class *Collections*

```
public void sort( List<E>)
public void sort( List<E>, Comparator<E>)
public void shuffle( List<E>, Comparator<E>)
```