I2206 – Data Structures Final Project
Name: Jean-Paul Khawam
File Number: 57300

Problem 1: Dynamic Arrays (Vectors)

Sample Input: *Two vectors, A and B*

Sample output:

```
Creating an empty array a...
Creating an array B with size 15, and initializing all values to 7...

Displaying array A: Array A is empty.
Displaying array B: 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7

Assigning the first 2 elements of A to 9...
Displaying array A: 9 9
Assigning the first 7 elements of B to 0...
Displaying array B: 0 0 0 0 0 0 0 7 7 7 7 7 7 7 7

Current size of array B: 15
Current number of elements in B: 15
Resizing B to 5...
Current size of array B: 5
Displaying array B: 0 0 0 0 0

Pushing back the number 88 to the array B
Inserting the value 44 at position 0...
Displaying array B: 44 0 0 0 0 0 88
Erasing the element at position 2...
The first element of B is 44
The last element of B is 88
Deleting the last element...
Displaying array B: 44 0 0 0 0
```

Problem 2: Queues

Sample Input:

```
Number of queues: 3

How many rounds would you like to run? 3
```

Sample Output:

```
Round 1:
Numbers inputted into each queue: 3  4  10
Amount of data outputted from the system: 7
Size of queues after output:
q0.size(): 0
q1.size(): 0
q2.size(): 10

Round 2:
Numbers inputted into each queue: 7  3  1
Amount of data outputted from the system: 19
Size of queues after output:
q0.size(): 0
q1.size(): 0
q2.size(): 2

Round 3:
Numbers inputted into each queue: 9  1  6
Amount of data outputted from the system: 8
Size of queues after output:
q0.size(): 1
q1.size(): 1
q2.size(): 8

Average size of queues:
q0: 0.33333
q1: 0.33333
q2: 6.66667
```

The ratio between ni and n should be that

$$n * 2 \geq \sum \mathrm{n_i}$$

so that ni doesn’t start to increase uncontrollably. This makes sure n always dominates it.

Problem 3: Stacks

Sample Input:

```
{8 6 4 1 1 7 1 4 1 3 7 1 4 4 4}
```

Sample Output:

```
Initial vector: 8 6 4 1 1 7 1 4 1 3 7 1 4 4 4
Sorted vector: 8 7 7 6 4 4 4 4 4 3 1 1 1 1 1

Stack 0: 8 1 1
Stack 1: 7 3
Stack 2: 7
Stack 3: 6 4
Stack 4: 4 4 1 1
Stack 5: 4 4 1
```

Complexity of the function: O(N logN), where N is the number of elements in the vector.

The complexities are from

1) The standard C++ sort function which has O(N logN) complexity
2) A for loop that runs N times, in it a for loop that runs as many times as there exists stacks. In the worst-case scenario (if all elements of the vector were 10), this would amount to a O(N^2) complexity. But the actual run-time is way less than that and runs around O(N logN).

The total complexity becomes O(N logN) + O(N logN) = O(N logN)

Problem 4: Trees

Sample Input: [TEXT FILE]

```
15

2 4 6 8 0 10 0 12 0 13 0 14 0 0 0
3 5 7 9 0 11 0 0 0 0 0 15 0 0 0

1 2 4 12 8 12 13 0
2 12 6 7 5 9 6 0
```

Sample Output:

```
1 and 2 are descendant-1
2 and 12 are descendant-3
4 and 6 are cousin2-2-1
12 and 7 are cousin4-2-1
8 and 5 are cousin2-1-2
12 and 9 are cousin2-1-4
13 and 6 are descendant-2
```

Complexity of the solution:

findPath() has a worst case of the node wanted being the last, i.e. O(n), and an average case of O(log N)

findDirectRelation() uses an existing vector, and has an average/almost always complexity of O(log N) (position of the node in the tree). The worst case is O(n) (When the two elements are the last two elements in the tree).

findCommonAncestor() also uses a vector, and has a worst case of O(log N), but can be way less than that. It really depends of where the ancestor is. The worst case is always less than O(n).

Total complexity:

Average/Normal Case: O(logN)+O(logN)+O(logN) = O(logN)

Worst Case: O(n)

Problem 5: Sorting

1) Sample Input: `{ 0,1,1,0,0,2,2,2,0,0,1,0,2,1 }`

   Sample output:
   ```
   Initial array: 0 1 1 0 0 2 2 2 0 0 1 0 2 1
   Sorted Array: 0 0 0 0 0 0 1 1 1 1 2 2 2 2
   ```

2) Sample Input: `{ 1,5,2,5,7,2,4,8,11,22,13,16,2,6,9,5,3,1 }`

   Sample Output: `1 1 2 2 2 3 4 5 5 5 6 7 8 9 11 13 16 22`

   Complexity: The complexity of the function is the complexity of passing once on each element of the input array θ(n) and then, for each occurrence of the value 0<i<27 insert it in the output array. Since we have n elements, this only takes θ(n) as well. The resulting complexity is θ(2n), or O(n).

3a) Sample Input: `{ "cab", "bcd", "axz", "xwy", "mpo", "dcv" }`

   Sample Output: `axz bcd cab dcv mpo xwy`

   Complexity: The complexity is identical to quicksort O(nlogn)

3b) Sample Input: `{ "climate", "justify", "bracket", "fortune", "dignity", "routine", "unaware", "popular", "royalty", "paradox", "absence", "hostage", "factory", "abolish", "compose" }`

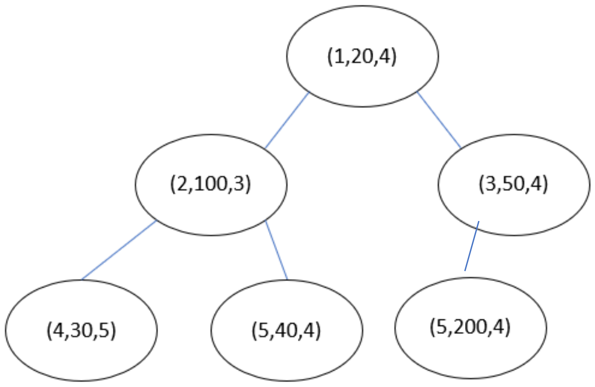
Sample Output: `abolish absence bracket climate compose dignity factory fortune hostage justify paradox popular routine royalty unaware`

Explanation/Reasoning: Radix sort is a sorting algorithm that repeatedly does a count sort based on the i-th digit in a number. We can modify that to count sort the i-th letter in a word. Moreover, it is even better because in normal radix sort you have to traverse the array once to get the maximum amount of digits, here we already know the fixed size m.

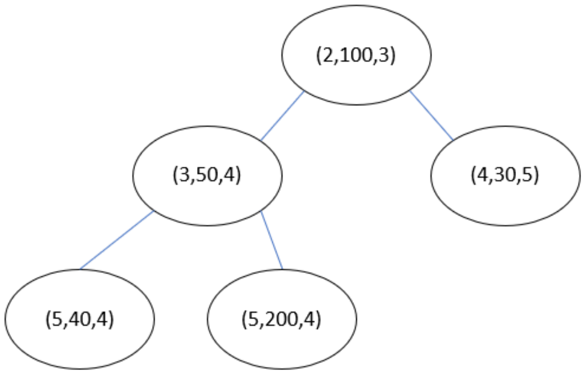
Complexity: The complexity of this radix sort is the complexity of doing count sort m times on n elements. One count sort on n elements is θ(3n) or O(n). So the total becomes m* θ(3n) = θ(3mn), or m*O(n) = O(mn)

Problem 6: Heaps

Initial heap: (ordered by priority)



Heap after first task is done:



Sample Input/Output:

```
Enter an id of a task: (20/30/40/50/100/200)
30
Your ID is: 30. The necessary time to print your task is: 14
```

Complexity of the function:

The complexity of the function calculateDuration() is the complexity of the first loop (which runs until it finds the element desired in the array.). This has as a worst case a linear complexity of O(n) if it is the last element. Plus the complexity of the second loop, which visits all the elements of the array once (O(n)). Hence the solution also has a complexity of O(n).

Problem 7: Hash Tables

Sample Input:

[TEXT FILE] **dictionary.txt :**

```
glue qualify porter abandon driver oppose fountain quote loss morale
bait stimulation
overall overeat wire produce way talkative dangerous land parameter
mail fur film name sanctuary end leader present stake retire fax
scheme attract cooperate be choice tense medal familiar news pocket
poison tap sum trip opinion reflect door take
```

[TEXT FILE] **paragraph.txt :**

```
glue constant module skate porter abandon tape slide detail fountain
representative acute
prevent bait stimulation overall overeat wire produce deputy try agree
end accompany enter swallow fax scheme attract cooperate be choice
contrary news pocket conceive opinion reflect door take
```

Sample Output:

```
constant
module
skate
tape
slide
detail
representative
acute
prevent
deputy
try
agree
accompany
enter
swallow
contrary
conceive
```

Complexity of the program:

Insertion: O(n) (each insert is O(1))

Output: to check if each word exists O(n).

Total complexity: O(n)