Task 2.2C
Data Structures and Algorithms

Question 1

First provide your student number here: ___ C23020001@cicra.edu.lk ____
Now write your final personalized algorithms out in full.

No	Personalized Algorithm	Operation count of worst case per line

i.	int count = 0;
	for (int i = 0; i < N; i++) {
	    if (rand() < 0.5) {
	        count++;
	        count--;
	        count++;
	    }
	}
	if (rand() > 0.5) {
	    int j = count;
	    while(j > 0) {
	        for (int k = N; k > N/2; k--) {
	        count++; }
	        j- -;
	    }
	}
		Line 1: Initialization (1 op)
		Line 2 : Condition check (N+1 ops), Increment (N ops)
		Line 3 : Random number generation and comparison (N ops)
		Line 4: Increment (c1 * N ops)
		Line 5: Decrement (c1 * N ops)
		Line 6: Increment (c1 * N ops)
		
		Line 7: Random number generation and comparison (1 op)
		Line 8: Assignment (1 op)
		Line 9: Condition check (j+1 ops)
		Line 10: Initialization (1 op), Condition check (N/2 + 1 ops), Decrement (N/2 ops)
		Line 11: Increment ((N/2 + 1) * j ops)
		Line 12: Decrement (j ops)

ii.	int count = 0;
	for (int i = 0; i < N-1; i++){
	    for (int j = i+1; j < N; j++ }
	        for (int k = N; k > N/2; k--) {
	        count++; }
	    }
	}
	int j = 0;
	int num = count;
	while(j < num) {
	    for (int k = N; k > N/2; k--){
	    count++; }
	    j++;
	}
		Line 1: Initialization (1 op)
		Line 2: Initialization (1 op), Condition check ((N-1) + 1 ops), Increment (N-1 ops)
		Line 3: Initialization ((N-1) ops), Condition check (sum from 2 to N-1 ops), Increment (sum from 2 to N-1 ops)
		Line 4: Initialization ((sum from 2 to N-1) ops), Condition check ((N/2 + 1) * (sum from 2 to N-1) ops), Decrement ((N/2) * (sum from 2 to N-1) ops)
		Line 5: Increment ((N/2 + 1) * (sum from 2 to N-1) ops) }
		Line 6: Initialization (1 op)
		Line 7: Assignment (1 op)
		line 8: Condition check (num + 1 ops)
		Line 9: Initialization (num ops), Condition check ((N/2 + 1) * num ops), Decrement ((N/2) * num ops)
		Line 10: Increment ((N/2 + 1) * num ops)
		Line 11: Increment (num ops)

Algorithm 1
N: `2N + 2` operations in a loop. Random check: `N` action inside the loop. Within the loop, there are `3*(c1 *N) ` operations for increment and decrement (or about `1.5N` operations for c1=0.5). Random inspection beyond the loop: `1` actions. Outside of the loop assignment: ` 1` operation. Condition check: `j +` operations while in a loop. `N/2 + 1 *j` operations are inner increment operation. ` j `

When these are added together, the total number of operation is
$2N + 2 + N + 1.5N + 1 + 1 + (j+1) + 2 * \left(\frac{N}{2} + 1\right) * j + j$
To further simplify, let's assume that `j` is originally set to `count`. For a high N, the largest terms would dominate the complexity. One can approximate the total as O (N^2).

Best Case:

- Best case occurs when `rand() < 0.5` and `rand() > 0.5` are never true.
- Operation count: 1 + (N + 1) + 1 = N + 3

Average Case:

- Average case assumes `rand() <0.5` and `rand() > 0.5` are true 50% of the time.
- Operation count: $1+(N+1)+N+\left(\frac{j*(\frac{N}{2})}{2}\right)= 1+N+1+N+\left(\frac{j*(\frac{N}{2})}{2}\right)$

Worst Case:

- Worst case occurs when `rand() <0.5` is true for all iterations and `read()>0.5` is true with `j=count=N`
- Operation count: $1+(N+1)+N*3+1+\frac{N*(\frac{N}{2})}{2}=1+N+1+3N+1+\frac{N*(\frac{N}{2})}{2}$

Algorithm 2

The code initialize `count` to zero to zero, taking 1 operation. The first `for` loop runs from 0 to N – 2. generating 2N operations. The second nested `for` loop runs from i + 1 to N – 1 for each iteration of the outer loop, resulting in approximately $\left(\frac{N(N-1}{2}\right)$ operations. Inside this loop, the third nested `for` loop runs from N down to N/2 adding $\left(\frac{N}{2}\right)$ operations per iteration of the second loop. This gives a total of $\left(\frac{N3}{4} - \frac{N2}{4}\right)$operations Each iteration of the innermost loop increments `count`, matching the operation of the third loop.

Next, `j` is initialized to zero, and `count` is assigned to `num`, each requiring 1 operation. The `while` loop runs `num` times, where `num` is approximately $\left(\frac{N3}{4}\right)$. Within the `while` loop, another `for` loop runs from N down N/2, adding $\left(\frac{N}{2}\right)$. Num operation. Incrementing `count` inside this loop adds another$\left(\frac{N}{2}\right)$. Num operations. Additionally, the `while` loop condition and increment of `j` add `num + 1` and `num` operations, respectively.

Summarizing the total operations are dominated by the largest terms, resulting in an overall complexity of O(N^3).

Best Case:

- Best case occurs when `N` is small, minimal loop executions.
- Operation count: 1+2(N-1)+1+1=2N+1

Average Case:

- Average case assumes `N` is moderate.
- Operation Count: 1+(N-1)+ $\left(\frac{N(N-1}{2}\right)$+$\left(\frac{N(N-1N/2}{2}\right)$+1+num(N/2)=1+(N-1)+ $\left(\frac{N(N-1}{2}\right)$+$\left(\frac{N(N-1N/2}{2}\right)$+1+$\left(\frac{num(\frac{N}{2})}{2}\right)$

Worst Case:

- Worst case occurs when `N` is large, with nested loops running fully.
- Operation count: 1+2(N-1)+N(N-1)/2+(N-1)(N/2)+(N-1)(N/2)+1+2num+1=1+2(N-1)+N(N-1)/2+(N-1)(N/2)+(N-1)(N/2)=1=2num=1

Question 2

Task 1.1P asked you to develop / provided you with a number of the Vector class's methods (and properties), such as Count, Capacity, Add, IndexOf, Insert, Clear, Contains, Remove, and RemoveAt. What is the algorithmic complexity of each of these operations? Does your implementation match the complexity of the corresponding operations offered by Microsoft .Net Framework for its List collection?

1. Count
Complexity: It is an O(1) operation if `count` is implemented as a property that only returns a field.

Comparison with List<T>: Because the `count` property returns the number of elements, this is likewise an O(1) operation

2. Capacity
Complexity: The implementation of `Capacity` as a property that yields the internal array's size results in an O(1) operation.

Comparison with List<T>: Furthermore, this is an O(1)process.

3. Add
Complexity: This process expands the end of the vector by one element. If the vector is full, ExtendData() is used to increase the capacity. Resizing the array has an O(n) time complexity because elements must be copied; however, the amortized time complexity is just O(1) because resizing is rare (doubling the size).

Comparison with List<T>: Moreover, List's Add operation has an amortized temporal complexity of O(1).

4. IndexOf
Complexity: Since this method searches the vector linearly for the element, its temporal complexity is O(n).

Comparison: For every scenario, the IndexOf operation on the List has an O(n) time complexity.

5. Insert:
Complexity: With this technique, all elements after an element are moved by one position by inserting it at a particular index. If the vector is full, it will enlarge and cost an additional O(n). The temporal complexity is O(n) for resizing and O(n) for shifting elements. When combined, these variables result in an O(n) total time complexity.

Comparison: The Insert function in a list has a temporal complexity of O(n) due to member relocation and potential resizing.

6. Clear:
Complexity: To reset the vector, this function makes a new array and sets Count to 0. This procedure has an O(1) time complexity because it does not loop over the objects.

Comparison: Similarly, the List's Clear operation has an O(1) temporal complexity.

7. Contains:
Complexity: The presence of an element is ascertained using this method by using IndexOf. Given the O(n) time complexity of IndexOf, the time complexity of Contains is likewise O(n).

Comparison: In the same way, the temporal complexity of the Contains method on a list is O(n).

8. Remove:
Complexity: This method first determines the element's index and then uses RemoveAt to remove it. Because shifting items add complexity, the element's complexity is O(n) to discover using IndexOf and O(n) to remove. Consequently, the total complexity to locate and eliminate the element is O(n).

Comparison: The Remove method in List has a temporal complexity of O(n).

9. RemoveAt:
Complexity: By using this technique, all items after it are shifted to fill the void left by removing the element at the specified index. This shifting operation has an O(n) temporal complexity (t).

Comparison: List's EliminationThe complexity of the at method is also O(n) in time..

10. ToString:
Complexity: This approach's temporal complexity is O(n), as iterating over the vector creates a string representation.

Comparison: List lacks a direct ToString method, but it would still have O(n) complexity if similar operations were iterated across the list.

Question 3

To address this problem, we need to identify a function f(n) that satisfies the following criteria:

1. f(n) is neither in Θ(n) nor in Θ(n^2): The growth of f(n) is not precisely at the pace of n or n^2.
2. f(n) is in O(n^2): In the top bound, f(n) expands at most as quickly as n^2.
3. f(n) can be represented with Θ: Although f(n) = Θ(g(n)), the function can still be asymptotically defined by some g(n).
4. f(n) is in Ω(n): In lower bound, f(n) grows at least as quickly as n.

Example function:

f(n) = n logn

Verifying the condition

1. f(n) is neither in θ(n) nor in θ(n^2):
   - Since the log n term causes n log n to expand faster than n, f(n) is not in θ(n).
   - Due to n log n growing slower than n^2, f(n) is not in (n^2). The growth rate of f(n) would have to match that of n^2 for it to be in θ(n^2). However, this is not the case.

2. f(n) is in O(n^2):
   - We must prove that the growth of f(n) = n logn is not greater than n^2.
   - As log n grows more slowly than n, log n logn will also increase more slowly than n^2.

3. f(n) can be represented with θ:
   - f(n) can be expressed as θ(n log n) as its asymptotic growth matches that of n log n.

4. f(n) is in Ω(n):
   - We prove that f(n) = n log n increases at least as rapidly as n to illustrate this.
   - Log n will always be bigger than n for sufficiently large n, because log n is positive for n > 1 and grows as n increases.
   - Consequently, for big n, there is a constant c > 0 such that
     - n log n >= c . n
   - f(n) = n log n is hence in Ω(n).

Merge Sort is a well-known algorithm that occasionally exhibits this complexity:

- best-case complexity: O(n log n)
- average-case complexity: O(n log n)
- worst-case complexity: O(n log n)

After sorting the array recursively in half, the array is merged using the Merge Sort method. Because the array is divided into half log n times and the merging process has a linear time complexity of O(n), the time complexity of the merge sort is O(n log n).

Heapsort is an additional example, with an average and worst-case time complexity of O(n log n).In the best, worst, and average circumstances, both of these sorting algorithms satisfy the Θ(n log n) criterion.

Question 4

a.

f(n) = $n^{1/2}$

g(n) = log n

$$\lim_{n\to\infty}\left(n^{1/2}\right)/logn \quad = \infty$$

F grows faster than g

f ∈ Ω(g)

b.

f(n) = 1500

g(n) = 2

$$\lim_{n\to\infty} 1500/2 = 750$$

f is 750 times bigger than g, they grow at the same rate (f = 750g)

f ∈ θ(g)

c.

f(n) = 800 x (2^n)

g(n) = 3^n

$$\lim_{n\to\infty} 800 \text{ x } (2^n) \;/\; 3^n = 0$$

f grows slower than g

f ∈ O(g)

d.

f(n) = 4^{n+13}

g(n) = 2^{2n+2}

$$\lim_{n\to\infty} 4^{n+13} / 2^{2n+2} = 16777216$$

f is 16777216 bigger than g, they grows at the same rate (f(n) = 16777216 * g(n))

f ∈ θ(g)

e.

f(n) = 9n x log n

g(n) = n x log n

$$\lim_{n\to\infty} \text{9n x log n /n x log n} = 9$$

f is 9 time bigger than g, they grows at the same rate (f(n) = 9 * g(n))

f ∈ θ(g)

f.

f(n) = n!

g(n) = (n + 1)!

$$\lim_{n\to\infty} n! /(\text{n} + 1)! = 0$$

F grows slower than g

f ∈ O(g)