Deakin University

Data Structures and Algorithms

OnTrack Submission

Basic questions on algorithmic complexity

Submitted By:
Jasveena Jasveena
s224001588
2025/04/02 02:51

Tutor:
Pooja Pancholi

Outcome	Weight
Document solutions	♦◊◊◊◊

great task to start learning about algorithms and their working

April 2, 2025

.TASK 2.2P

Basic questions on algorithmic complexity

SUBMITTED BY – JASVEENA -224001588

1. Study the Introspective Sort, which is a default sorting algorithm implemented in the Array.Sort method. You must address the following questions: –

A. What is special about the Introspective Sort?
   - Introspective Sort (Introsort) is a hybrid sorting algorithm used in .NET's Array.Sort method. Introsort is special because:
   - It starts as QuickSort, which is typically very fast for most cases.
   - If QuickSort recursion depth exceeds a certain limit, it switches to HeapSort to prevent worst-case performance ($O(n^2)$) which is the slowest
   - For small arrays, it applies Insertion Sort, which is efficient for nearly sorted data.
   - This adaptive approach ensures both average-case efficiency and worst-case performance guarantees.

B. Does this method result in an unstable sorting; that is, if two elements are seen equal, might their order be not preserved after the data collection is sorted? Or does it perform a stable sorting, which preserves the order of elements that are seen equal?

   - Introsort is an unstable sorting algorithm.
   - This means that if two elements are equal, their original order may change after sorting.
   - It uses QuickSort and HeapSort, which do not guarantee order preservation for equal elements.
   - QuickSort swaps elements based on pivots, which can change positions of equal elements.
   - HeapSort moves elements in a binary heap, breaking original ordering.

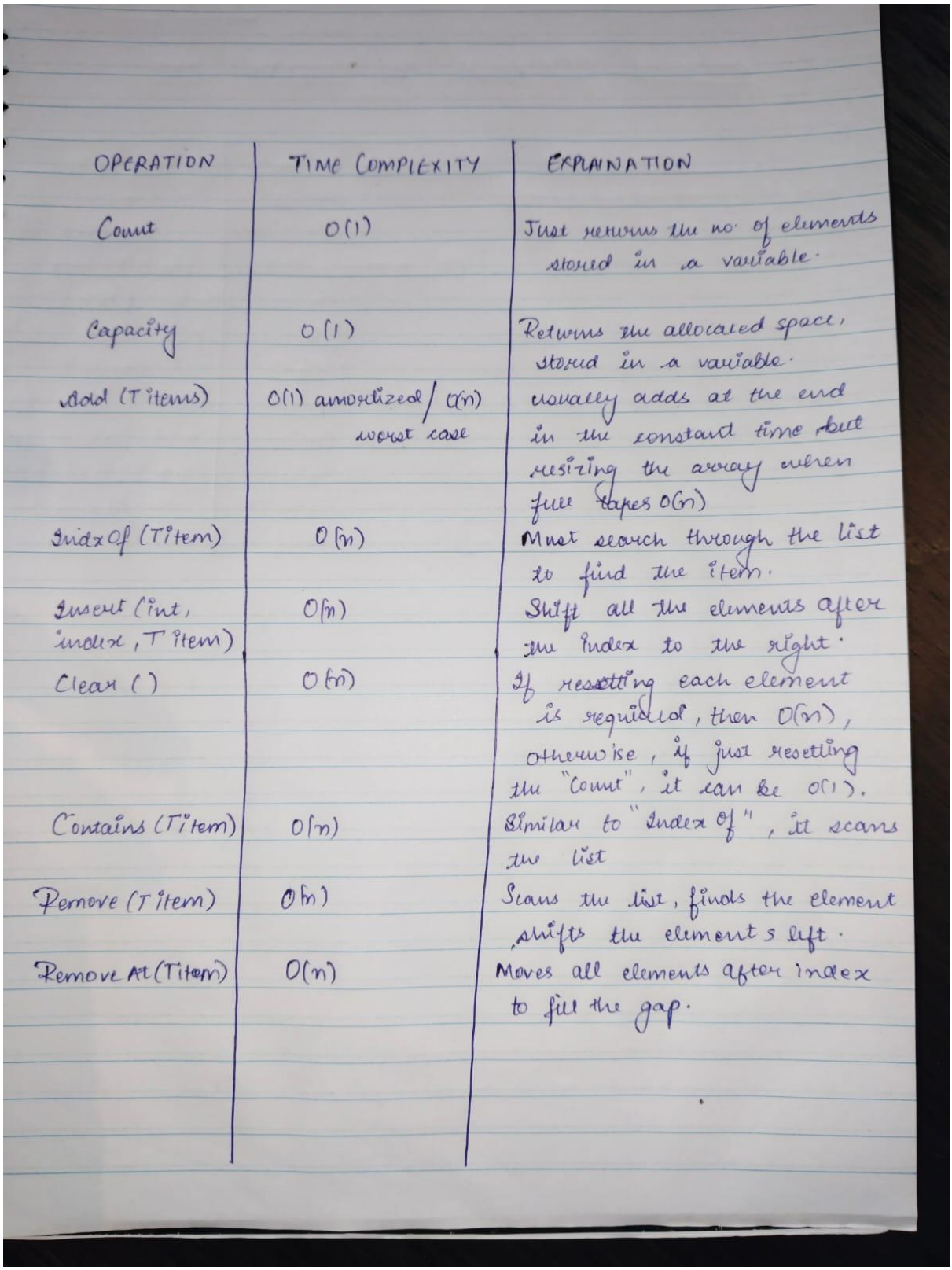
C. What is the time complexity of this sorting algorithm?
   Time complexity tells us **how fast an algorithm works** when sorting more and more data. In this case , introsort works well in most cases due to insertion sort , quicksort and heapsort.

If the data is already **almost sorted**, Introsort uses **Insertion Sort**, which is very quick.

In average number of cases **most of the time** when sorting random numbers, Introsort mainly uses **QuickSort**, which works well in most cases.

In some worst case scenarios , Some sorting methods (like QuickSort) **can slow down badly** in the worst case. To **avoid** this, Introsort **switches to HeapSort**, which keeps it efficient.

2. In Task 1.1P on the Vector class, you have completed/been provided with a number of methods (and properties), such as Count, Capacity, Add, IndexOf, Insert, Clear, Contains, Remove, and RemoveAt. Answer the following questions:
   – What is the time complexity of each of these operations?



OPERATION	TIME COMPLEXITY	EXPLANATION
Count	O(1)	Just returns the no. of elements stored in a variable.
Capacity	O(1)	Returns the allocated space, stored in a variable.
Add (T item)	O(1) amortized / O(n) worst case	usually adds at the end in the constant time but resizing the array when full takes O(n)
IndexOf (T item)	O(n)	Must search through the list to find the item.
Insert (int, index, T item)	O(n)	Shift all the elements after the index to the right.
Clear ()	O(n)	If resetting each element is required, then O(n), otherwise, if just resetting the "Count", it can be O(1).
Contains (T item)	O(n)	Similar to "IndexOf", it scans the list
Remove (T item)	O(n)	Scans the list, finds the element, shifts the elements left.
RemoveAt (T item)	O(n)	Moves all elements after index to fill the gap.

Where ,
O(1) (Constant Time)
O(1) Amortized (Constant Time on Average, but Sometimes Slower)

O(n) (Linear Time)

– Does your implementation match the complexity of the equivalent operations provided by the Microsoft .NET Framework for its List collection?

- .NET's List<T> has the **same complexities** as Vector<T> because they both use **dynamic arrays** internally.
- However, **Microsoft's implementation is highly optimized**, using methods like **Array.Copy** for shifting elements, which makes operations slightly faster in practice.

3. Answer and explain whether the following statements are right or wrong.
   3.a. A θn^3 algorithm always takes longer to run than a $\theta(\log n)$ algorithm.
   Incorrect –
   Θ(n) is not always slower than Θ(log n).
   Time complexity describes **growth rate**, not exact speed.
   For **small n**, an O(n) algorithm can be **faster** than an O(log n) one due to constant factors.

   3.b. The best-case time complexity of the Bubble Sort algorithm is always On .
   Correct – but only if bubble sort is optimized
   Otherwise it will run through all passes and make the best case 0(n^2) instead which is much slower and plus the worst case.

   3.c. The worst-case time complexity of the Insertion Sort algorithm is always On^2 .
   correct –
   the worst-case time complexity of Insertion Sort is always **O(n²)**. This occurs when the input array is sorted in reverse order, requiring each element to be compared and shifted for every insertion.

   3.d. The Selection Sort is an in-place sorting algorithm.
   Correct –
   Selection Sort does not use extra memory aside from a few variables.
   It swaps elements within the array, making it in-place.

   3.e. The worst-case space complexity of the Insertion Sort algorithm is $O(1)$.

   Correct – Insertion Sort sorts the array without extra storage, using only a few temporary variables.

   3.f. 2 +2 = $O(1)$
   Correct- Since 2+2 = 4 a constant ,
   In asymptotic notation, O(1) represents a constant time or value that doesn't grow with the input size.

3.g. n^3 +10^6n = O(n^3)
Correct - In Big-O notation, we focus on the **highest-order term** and ignore **constants** because they have negligible impact for large n

3.h. nlog n = O (n)
Incorrect – n log n grows faster than n for large n., it could have been n log n = O(n log n)

3.i. logn = o(n)
Correct - ittle-o (o) means strictly slower.
Since log n grows much slower than n, this is correct.

3.j. log n + 2^100 = $\Theta(log$n)
correct –
log n is the dominant term because it grows with **n**. $\mathbf{2^{100}}$ is a constant and does not depend on **n**. In asymptotic notation, constants are ignored when determining the growth rate.
Since **Θ(log n)** describes functions that grow at the same rate as **log n**, the equation holds true.

3.k. n log n = Ω (n)
Correct - Ω(n) means "grows at least as fast as n."
Since n log n grows faster than n, this is true.

3.l. n +2^n = O (n)
Incorrect –
n grows linearly.
2^n grows exponentially and dominates n for large values of n.

3.m. $n + 100\mathrm{n}/\log \mathrm{n} = \mathrm{o}\ (\mathrm{n})$
Incorrect-
Little-o (o) notation means *strictly slower* (i.e., it must grow strictly slower than n).
The first term n grows linearly.
The second term is slightly smaller than n but still asymptotically close to n.
The dominant term is n itself.

3.n. $\mathrm{n}! = \mathrm{O}(\mathrm{n})$
Incorrect-
Factorial growth n! is much faster than linear growth n:

- Example:
  - For n=5:5! =120, while O(n) is just 5.
  - For n=10:10!= 3628800, while O(n) is just 10.
  - For large n, n! completely dominates n.

4. 4. Give your best- and worst-case asymptotic runtime analysis to the following code snippets.

4.1. Let flag be a random Boolean variable, whose value is either true or false.

```
int count = 0;
for (int i = 0; i < n; i++)
{
   if (flag)
   {
        for (j = i+1; i < n; j++)
        {
             count = count + i + j;
        }
   }
}
```

Best-case: O(n)

When flag is false, inner loop doesn't run; outer loop runs n times.

Worst-case: O(n²)

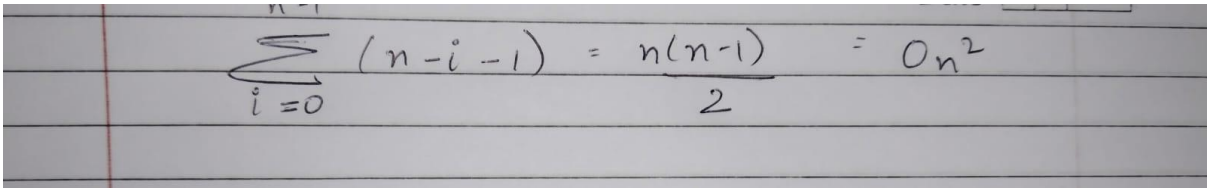
When flag is true, inner loop runs for each i, totaling ~n(n-1)/2 iterations.

Why ?

The **outer loop** runs n times.

The **inner loop** runs from i+1 to n, so it executes (n - i - 1) times.

The total number of operations if the inner loop executes every time (i.e., if flag == true) is:



4.2 Let random() be a function producing a real random number in the range [0,1).

```
int count = 0;
for (int i = 0; i < n; i++)
{
   int num = random();
   if( num < 0.01 )
   {
       count = count + 1;
   }
}
int num = count;
for (int j = 0; j < num; j++)
{
   count = count + j;
}
```

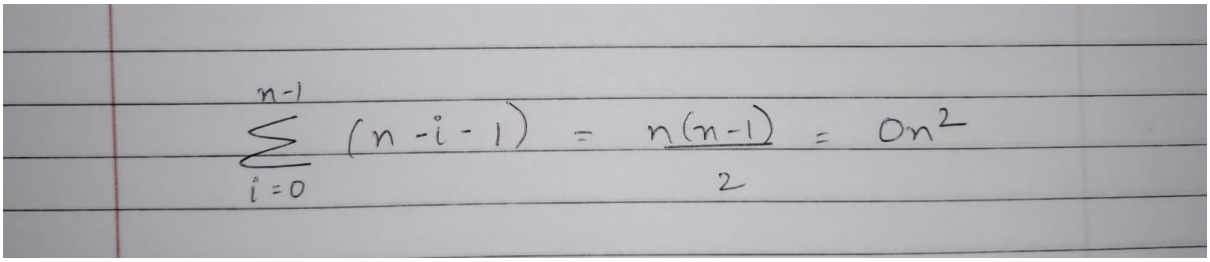
Best-case: O(n)
When count is 0 (no num < 0.01), second loop doesn't run; first loop is O(n).
Worst-case: O(n²)
When count is n (all num < 0.01), second loop runs n times, making it O(n²).
Why ?
The **first loop** always runs n times: O(n).
The condition num < 0.01 happens very rarely.
So count is expected to be around 0.01n in average case.

4.3 Let Compare(x, y) be a method with the time complexity of $\Theta 1$.

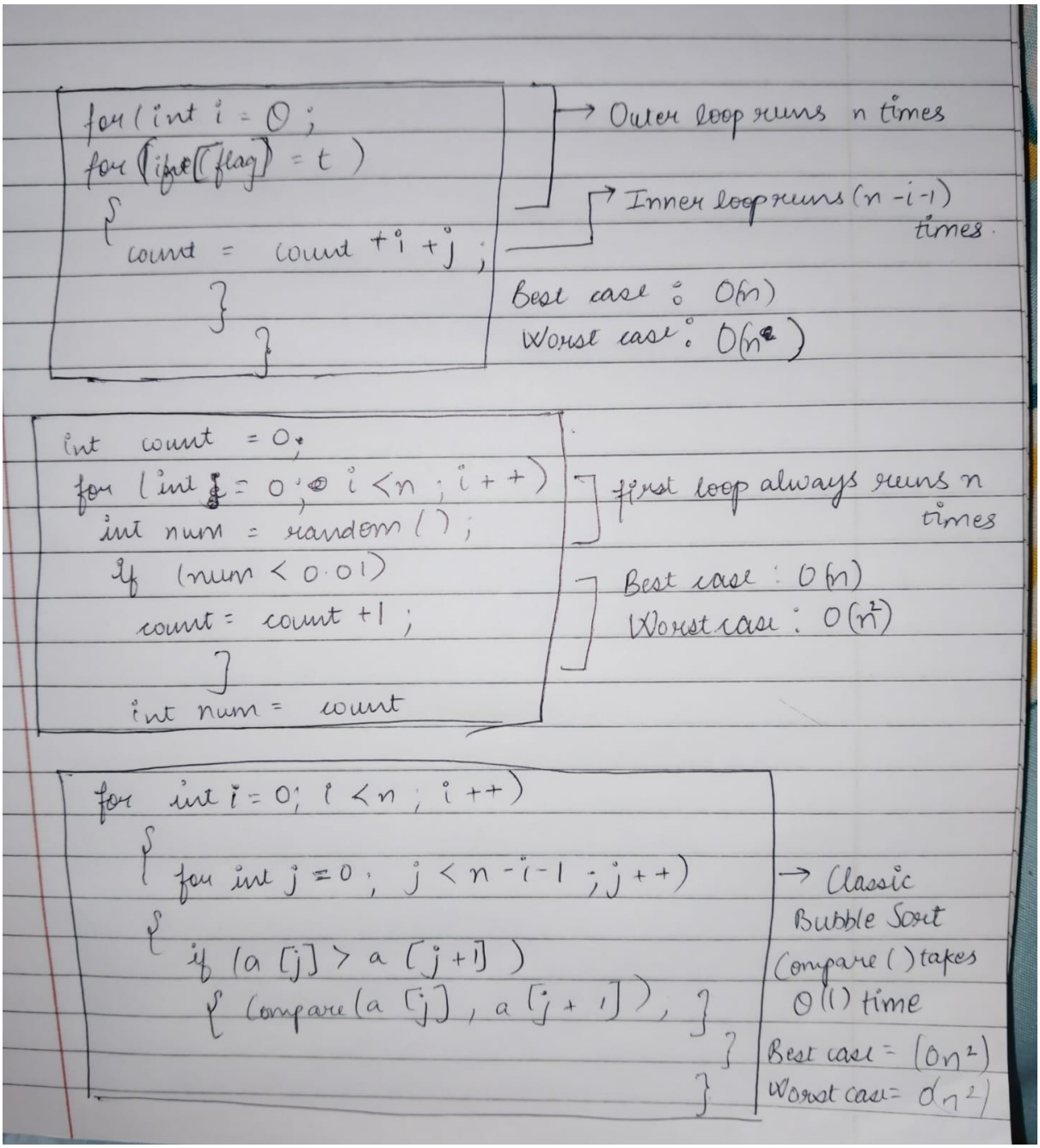
```
for (int i = 0; i < n; i++)
{
   for (int j = 0; j < n-i-1; j++)
   {
       if (a[j] > a[j+1])
       {
           Compare(a[j], a[j + 1]);
       }
   }
}
```

Best-case: O(n²)
Even if no swaps occur (array sorted), nested loops run ~n(n-1)/2 times;
Compare is Θ(1).

Worst-case: $O(n^2)$
Same loop structure regardless of comparisons; ~n(n-1)/2 iterations, Compare is Θ(1).
Why ?
Follows a Classic Bubble Sort structure.
The outer loop runs n times and
The inner loop runs (n - i - 1) times.
Total comparisons:



Compare() takes Θ(1) time.


for (int i = 0;
for (if[flag] = t)
{
count = count + i + j;
}
}
→ Outer loop runs n times
→ Inner loop runs (n - i - 1) times.
Best case : O(n)
Worst case : O(n²)
int count = 0;
for (int i = 0; i < n; i++)
int num = random();
if (num < 0.01)
count = count + 1;
]
int num = count
first loop always runs n times
Best case : O(n)
Worst case : O(n²)
for int i = 0; i < n; i++)
{
for int j = 0; j < n - i - 1; j++)
{
if (a[j] > a[j+1])
{ compare(a[j], a[j+1]); }
}
}
→ Classic Bubble Sort
Compare() takes O(1) time
Best case = (On²)
Worst case = O(n²)