# Workshop 2: Analysis of Algorithms

# Questions

# Part A: Big-O Notation

1. Each of the following multiple choice questions has four (4) alternative answers. Choose one of the answers that best answers the question.
2. Which of the following big-O notations represents the computation time of the algorithm grows in **direct** proportion to the size of the input in the worst case?
   1. O(1)
   2. O(n)
   3. O(n2)
   4. O(2n)
3. Which of the following big-O notations represents the computation time of the algorithm does not change with the size of the input?
   1. O(1)
   2. O(n)
   3. O(n2)
   4. O(2n)
4. Which of the following big-O classes does the efficiency function belongs to?
   1. O(1)
   2. O(n)
   3. O(n2)
   4. O(n3)

# Part B: Theoretical Algorithm Analysis

## Identifying efficiency classes in big-O notation by inspection

For each of the pseudocode fragments below, **let *f*(*n*) be its exact efficiency function with respect to input value *n***, assuming that the **basic operation** of interest is the assignment statement ‘***x* ← *x* + 1’**.

For each of the following pseudocode fragments, **derive its efficiency function** and then **determine its time complexity in the worst scenario in big-O notation**.

1. **for** *j* ← 0 **to** *n* − 1 **do**

*x* ← *x* + 1

Big-O Class: O(n), since there is a **single** for loop.

Efficiency function: f(n) = n

1. **for** *i* ← 0 **to** *n* − 1 **do**

**for** *j* ← 0 **to** *n* − 1 **do**

*x* ← *x* + 1

Big-O Class: O(n2), since there is a **nested** for loop of depth 2.

Efficiency function: f(n) = n2

1. **if** *B*

**for** *j* ← 0 **to** *n* − 1 **do**

*x* ← *x* + 1

**else**

**for** *i* ← 0 **to** *n* − 1 **do**

**for** *j* ← 0 **to** *n* − 1 **do**

*x* ← *x* + 1

Big-O Class: O(n2)

Efficiency function:



If you need to calculate an average efficiency function



1. **for** *i* ← 0 **to** *n* − 1 **do**

*x* ← *x* + 1

**for** *i* ← 0 **to** *n* − 1 **do**

**for** *j* ← 0 **to** *n* − 1 **do**

*x* ← *x* + 1

Big-O Class: O(n2)

Efficiency function: f(n) = n + n2

1. *i* ← *n*

**while** *i* ≥ 1 **do**

*x* ← *x* + 1

*i* ← *i*/2

Big-O Class: O(log(n))

Efficiency function: f(n) = log2(n)

Optionally:

Start with n = 256

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| iterations | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| i | 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 | 0.5 |

256 = 28

8 = log2(256)

Number of iterations is approximately math.floor(log2(n + 1))

1. What is the worst-case time complexity in **big-O notation** for each of the following algorithms?
2. **ALGORITHM** *IsSorted*(*A*[0..*n* − 1])

// Returns true if all the numbers in an array *A* of length *n*

// are sorted in nondecreasing order, false otherwise.

**if** *n* ≤ 1

**return** true

**else**

**for** *i* ← 0 **to** *n* − 2 **do**

**if** *A*[*i*] > A[*i* + 1]

**return** false

**return** true

Big-O Class: O(n)

Best case scenario? When n = 1, or when we return false in the first iteration

Worse case? When the list is sorted in non-decreasing order

1. **ALGORITHM** *MinDistance*(*A*[0..*n* − 1])

// Returns the separation between the two closest numbers in a given array *A*

// of *n* numbers, or special value infinity (∞) if the array contains fewer than

// two numbers

*d* ← ∞

**for** *i* ← 0 **to** *n* − 2 **do**

**for** *j* ← *i* + 1 **to** *n* − 1 **do**

*d* ← min(*d*, |*A*[*i*] − *A*[*j*]|)

**return** *d*

Big-O Class: O(n2)

Challenge: find the efficiency function?

1. **ALGORITHM** *ReverseArray*(*A*[0..*n* − 1])

// Reverses the order of the items in an array *A* of length *n*.

**for** *i* ← 0 **to** ⎣*n*/2⎦ − 1 **do**

*temp* ← *A*[*i*]

*A*[*i*] ← *A*[*n* − 1 − *i*]

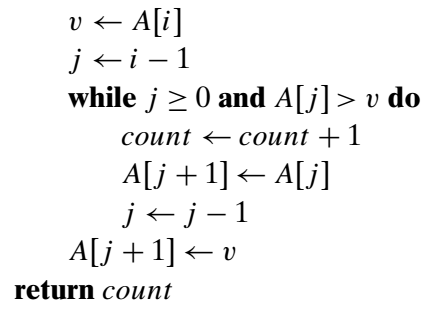
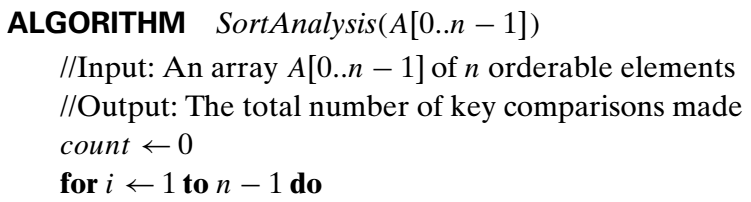
*A*[*n* − 1 − *i*] ← *temp*

Big-O Class: O(n)

What does ⎣*n*/2⎦ mean? Math.Floor(double a), rounds a number down to the nearest integer

## Part C: Empirical Algorithm Analysis

1. Consider the following well-known sorting algorithm, which is studied later in the book, with a counter inserted to count the number of key comparisons.



Is the comparison counter inserted in the right place? If you believe it is, prove it; if you believe it is not, make an appropriate correction.

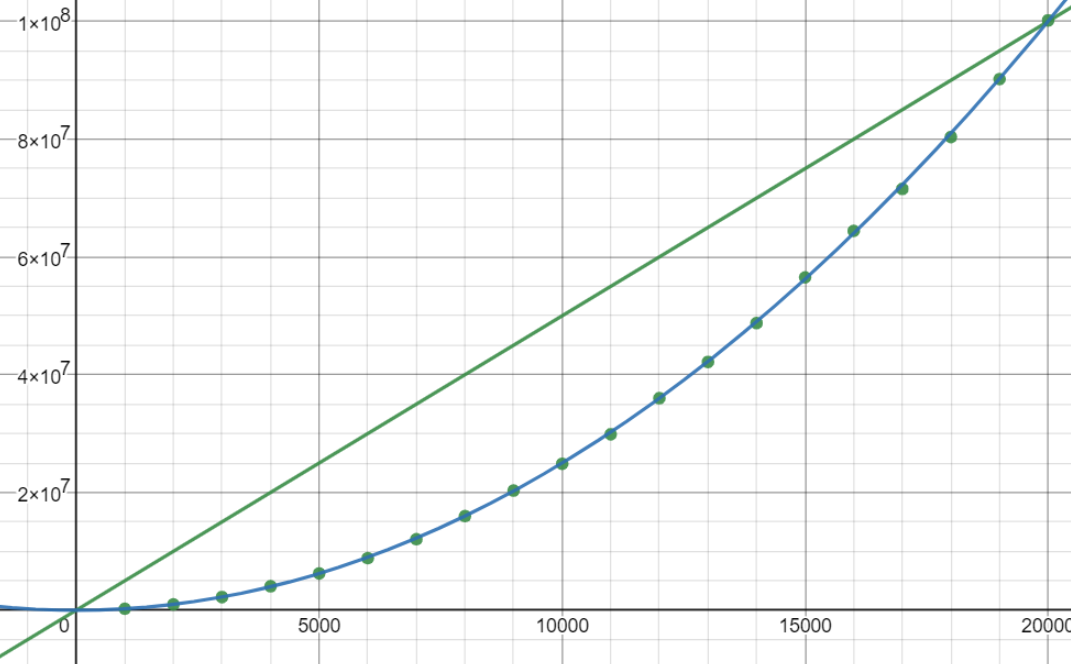
The comparison counter was not inserted in the right place, because when the first condition (j >= 0) of the while loop fails, the program does not actually checks the basic operation (the comparison A[j] > v).

To correct this, we add if (j < 0) count = count + 1; after the while loop.

1. Perform the following tasks one by one in the order:
   1. Run the program of Question 4, with a properly inserted counter (or counters) for the number of key comparisons, on 20 random arrays of sizes 1000, 2000, 3000, …, 20,000.

|  |  |
| --- | --- |
| **n** | **Number of comparisons** |
| 1000 | 253,995 |
| 2000 | 988,511 |
| 3000 | 2,235,906 |
| 4000 | 4,073,733 |
| 5000 | 6,258,177 |
| 6000 | 8,844,661 |
| 7000 | 12,070,944 |
| 8000 | 15,992,942 |
| 9000 | 20,331,408 |
| 10000 | 24,866,835 |
| 11000 | 29,882,651 |
| 12000 | 36,012,856 |
| 13000 | 42,162,286 |
| 14000 | 48,767,536 |
| 15000 | 56,533,761 |
| 16000 | 64,419,857 |
| 17000 | 71,545,356 |
| 18000 | 80,367,303 |
| 19000 | 90,201,648 |
| 20000 | 100,166,631 |

* 1. Analyse the data obtained to form a hypothesis about the algorithm’s average case efficiency.



Plotting number of comparisons against the input size show that there is a curve.

When we try to fit the data points with an O(n) efficiency function: f(n) = a n, we see that **it doesn’t fit** (a = 5000, or f(n) = 5000n). So the complexity is not O(n).

When try to fit the data with an O(n2) efficiency function: f2(n) = b n2, we see that letting **b = 0.25 fits almost perfectly** (based on trial and error)

We confirm that this algorithm is O(n2).

* 1. Estimate the number of key comparisons we should expect for a randomly generated array of size 25,000 sorted by the same algorithm.

Using the efficiency function found by trial and error, we found

f2(n) = 0.25n2

So for 25,000 inputs, f2(25000) = 0.25 \* (25000)2 = 156,250,000 comparisons

By forming a hypothesis f2(n) = bn2, we can find b by dividing f2(n) by n2.

|  |  |
| --- | --- |
| 20000 | 100,166,631 |

100,166,631 / (20,000)2= 0.2504

Or we say the efficiency function is around f2(n) = 0.2504n2