**Analyzing Selection Sort**

**SelectionSort(numbers, numbersSize) {**

**i = 0**

**j = 0**

**indexSmallest = 0**

**temp = 0 // Temporary variable for swap**

**for (i = 0; i < numbersSize - 1; ++i) {**

**// Find index of smallest remaining element**

**indexSmallest = i**

**for (j = i + 1; j < numbersSize; ++j) {**

**if ( numbers[j] < numbers[indexSmallest] ) {**

**indexSmallest = j**

**}**

**}**

**// Swap numbers[i] and numbers[indexSmallest]**

**temp = numbers[i]**

**numbers[i] = numbers[indexSmallest]**

**numbers[indexSmallest] = temp**

**}**

**}**

Assume that N = numbersSize

Inner for Loop:

|  |  |
| --- | --- |
| **ONCE** | **REPEAT** |
| **j = i + 1** | **if ( numbers[j] < numbers[indexSmallest])** |
| **j = numbersSize** | **indexSmallest = j**  **j < numbersSize**  **++j** |

The inner loop iterates (N – i – 1) times and each one has 4 instructions, to the number of instructions will be T(a) = 4 \* (N – i - 1) + 2 = 4N – 4i – 4+2= 4N-4i-2

Outer for Loop:

|  |  |
| --- | --- |
| **ONCE** | **REPEAT** |
| **i = 0** | **indexSmallest = i** |
| **i = numbersSize - 1** | **temp = numbers[i]**  **numbers[i] = numbers[indexSmallest]**  **numbers[indexSmallest] = temp**  **i < numbersSize - 1**  **++i**  **//inner for loop =** 4N-4i-2 |

The outer loop iterates numbersSize times, therefore the number of instructions is T(b) = (N-1) \* (4N – 4i - 2+ 6) + 2 = (N-1) \* (4N-4i+4) + 2 = 4N^2 – 4iN +16N – 4N + 4i – 4 + 2 = 4N^2 – N(4i-12) + 4i -2. So, the total number of instructions is T(N) = 4N^2 – N(4i-12) + 4i + 2, adding the initial 4 instructions at the beginning of the code.

That would lead to O(N^2), since all constants are dropped.

Other way of looking at time complexity is this:

Since this algorithm consists of two for loops. If the size of the array to be sorted is N then note that the outer loop executes N-1 times. The inner loop executes N-1, then N-2, then N-3, …, then 1 time, taking an average of N/2 executions, or N/2 comparisons.

Observe that the algorithm needs three instructions of constant time to perform the “swapping” of two elements at the end of every inner loop. Therefore, there is a total of (N - 1) \* (N/2) comparisons resulting with O(N^2) of time complexity.

The **average/best/worst case scenario** in this sorting algorithm is the same since the number of instructions executed depends strictly in the size of the array and not in how the array is sorted initially.

**Analyzing Insertion Sort**

**InsertionSort(numbers, numbersSize) {**

**i = 0**

**j = 0**

**temp = 0 // Temporary variable for swap**

**for (i = 1; i < numbersSize; ++i) {**

**j = i**

**// Insert numbers[i] into sorted part**

**// stopping once numbers[i] in correct position**

**while (j > 0 && numbers[j] < numbers[j - 1]) {**

**// Swap numbers[j] and numbers[j - 1]**

**temp = numbers[j]**

**numbers[j] = numbers[j - 1]**

**numbers[j - 1] = temp**

**--j**

**}**

**}**

**}**

Assume that N = numbersSize

Inner while Loop:

|  |  |
| --- | --- |
| **ONCE** | **REPEAT** |
| **It depends on the** | **Temp = numbers[j]** |
| **initial array** | **numbers[j] = numbers[j-1]**  **numbers[j-1] = temp**  **--j**  **j > 0**  **numbers[j] < numbers[j - 1])** |

The number of iterations the loop is executed depends on how many elements is numbers[j] less than numbers[j - 1] in the already sorted section of the array. So, assume that this number is “C”, where its minimum value is 0 and its maximum value is “i” (the number of elements in the sorted part of the array). Therefore, the number of instructions is T(a) = 6C

Outer for Loop:

|  |  |
| --- | --- |
| **ONCE** | **REPEAT** |
| **i = 1** | **j= i** |
| **i = numbersSize** | **//inner while loop** |

The outer loop iterates N-1 times, so the number of instructions in this loop will be T(b) = (N-1) \* (6C + 1) + 2. Note that there are 3 instructions out of the for loop.

This would yield to the total number of instructions in this sorting algorithm:

T(N) = (N-1) \* (6C + 1) + 5

**Best case:** This scenario depends strictly in C being always 0, therefore T(N) = N-1 + 5 and O(N) = N, this case would be when the array is already sorted, so there are no changes made.

**Worst case**: The worst-case scenario is when C = i, where i goes from 1 to N-1. This equation helps to find the total of operations needed:

**1 + 2 + 3 +….+ N-2 + N-1 =**

But since T(N) = 6C+6, we get that T(N) = 3(N-1)(N) + 6 and that O(N^2).

**Average case:** The inner loop executes an average of N/4 times, N/4 being the average of best and worst case, therefore we can substitute that with C. T(N) = (N-1) \* (6(N/4) + 1) + 5

T(N) = (N-1) \* (3N/2 + 1) + 5

T(N) = (3N^2/2) + N - 3N/2 + 4, therefore

O(N^2)

**Analyzing Bubble Sort**

**BubbleSort(numbers, numbersSize) {**

**for (int i = 0; i < numbersSize - 1; i++) {**

**for (int j = 0; j < numbersSize - i - 1; j++) {**

**if (numbers[j] > numbers[j+1]) {**

**temp = numbers[j]**

**numbers[j] = numbers[j + 1]**

**numbers[j + 1] = temp**

**}**

**}**

**}**

**}**

Assume that N = numbersSize

Inner for Loop:

|  |  |
| --- | --- |
| **ONCE** | **REPEAT** |
| **int j = 0** | **if (numbers[j] > numbers[j+1])**  **temp = numbers[j]** |
| **N – i - 1** | **numbers[j] = numbers[j+1]**  **numbers[j+1] = temp**  **j++**  **j < numbersSize – i - 1** |

The loop executes N–i–1 times, each with 6 instructions, therefore T(a) = 6\*(N-i-1)

Outer for Loop:

|  |  |
| --- | --- |
| **ONCE** | **REPEAT** |
| **i = 0** | **j= i** |
| **i = numbersSize-1** | **//inner while loop** |

The outer loop executes N-1 times, therefore T(b) = 6(N-1) \* (N-i-1)

T(N) = 6N^2 – 6Ni – 12N + 6i + 6. Note that i is in average N/2 so

T(N) = 6N^2 – 6N(N/2) – 12N + 6N/2 + 6

T(N) = 6N^2 – 3N^2 – 12N + 3N + 6

T(N) = 3N^2 – 9N + 6

O(N^2)

**Best case:** This scenario depends on the given array; the best case is achieved when the array is already sorted, then the inner loop will execute 3\*(N-i-1), because the if statement is never executed, so there is only three instructions inside each iteration of the inner loop. That’s why T(N) = 3N^2 – 3Ni – 6N + 3i + 3,

T(N) = 3N^2 – 3N(N/2) – 6N + 3(N/2) + 3

T(N) = 1.5N^2 – 4.5N + 3

and O(N^2).

**Worst case**: The worst-case scenario is when the array is sorted from maximum to minimum and therefore executing all 6 statements in the inner loop. Leading to an average of N/2 executions, therefore:

T(N) = 6(N-1)(N/2)

T(N) = 3(N-1)(N)

T(N) = 3N^2 – 3N

and O(N^2).

**Average case:** The average case is just the average of the best case and the worst case

T(N) = 2.25N^2 – 2.25N,

And O(N^2).

**Analyzing Binary Search**

**BinarySearch(numbers, numbersSize, key) {**

**mid = 0**

**low = 0**

**high = numbersSize - 1**

**while (high >= low) {**

**mid = (high + low) / 2**

**if (numbers[mid] == key) {**

**return mid**

**}**

**if (numbers[mid] < key) {**

**low = mid + 1**

**}**

**else if (numbers[mid] > key) {**

**high = mid - 1**

**}**

**}**

**return -1 // not found**

**}**

While loop

|  |  |
| --- | --- |
| **ONCE** | **REPEAT** |
|  | **(high >= low)**  **mid = (high + low) / 2** |
|  | **if (numbers[mid] == key)**  **return mid**  **if (numbers[mid] < key)**  **low = mid + 1**  **else if (numbers[mid] > key)**  **high = mid - 1** |

**Best case:** This occurs when the first element the algorithm selects is the element you were looking for; the least number of instructions would be T(N) = 7. And O(1).

**Worst case:** This case occurs when the element is not found, and the method return -1. This would take at most T(N) = 7\*logN + 4, leading to O(logN)

**Average case:** This is just the average of the vest case and the worst case, which would be

T(N) = 3.5logN + 5

O(logN)