David Schulz

CS 4981 021

4/1/2022

**Exercise 2: Big Oh**

**Math Function Big Oh**

1. f(x) = x2 + 4x + 4
   1. O(g(n)) = O(n2)
   2. M = 9
   3. n0 = 1
2. f(x) = 1/x2
   1. O(g(n)) = O(1)
   2. M = 1
   3. n0 = 1
3. f(x) = 3
   1. O(g(n)) = O(1)
   2. M = 3
   3. n0 = 0
4. f(x) = 12x + 1000
   1. O(g(n)) = O(n)
   2. M = 1012
   3. n0 = 1
5. f(x) = ex
   1. O(g(n)) = O(en)
   2. M = 1
   3. n0 = 0

**Interpreting Algorithms**

1. findIt1()
   1. Describe what method does
      1. Finds the smallest value in the input array.
   2. Determine the algorithmic complexity (Big Oh)
      1. O(n)
   3. Determine the space complexity (Big Oh)
      1. 1 array: O(n)
2. sumIt1()
   1. Describe what method does
      1. Calculates the sum of all values in the input array.
   2. Determine the algorithmic complexity (Big Oh)
      1. O(n)
   3. Determine the space complexity (Big Oh)
      1. 1 array: O(n)
3. allSum1()
   1. Describe what method does
      1. Calculates the sum of all of the values up to each index of the input array, so the output array is the same length as the input.
   2. Determine the algorithmic complexity (Big Oh)
      1. O(n2)
   3. Determine the space complexity (Big Oh)
      1. 2 arrays: f(2n) = O(n)
4. allSum2()
   1. Describe what method does
      1. Calculates the sum of the value before and at each index of the input array, so the output array is the same length as the input.
   2. Determine the algorithmic complexity (Big Oh)
      1. O(n)
   3. Determine the space complexity (Big Oh)
      1. 2 arrays: f(2n) = O(n)
5. sortIt1()
   1. Describe what method does
      1. Sorts the input array in increasing order using the selection sorting algorithm.
   2. Determine the algorithmic complexity (Big Oh)
      1. O(n2)
   3. Determine the space complexity (Big Oh)
      1. 1 array: O(n)
6. sortIt2()
   1. Describe what method does
      1. Sorts the input array in increasing order based on the number of occurrences of each value up to the maximum value in the input array.
   2. Determine the algorithmic complexity (Big Oh)
      1. 3 separate loops, iterations always ≤ n: f(3n) = O(n)
   3. Determine the space complexity (Big Oh)
      1. 2 arrays: f(2n) = O(n)

**Comparing Runtime with Java Library**

1. findIt1, libFindIt
   1. findIt1()

|  |  |  |
| --- | --- | --- |
| Elements | Runtime | Memory Usage |
| 10 | 1700 | 7 |
| 100 | 1300 | 7 |
| 1,000 | 10,500 | 7 |
| 10,000 | 97,000 | 7 |
| 100,000 | 801,000 | 7 |

* 1. libFindIt()

|  |  |  |
| --- | --- | --- |
| Elements | Runtime | Memory Usage |
| 10 | 36,837,700 | 10 |
| 100 | 30,100 | 10 |
| 1,000 | 100,000 | 10 |
| 10,000 | 229,800 | 10 |
| 100,000 | 1,375,400 | 10 |

* 1. Research about each method. Why is one faster/slower more/less a memory hog than the others?
     1. libFindIt uses an Array stream to find the minimum value, while findIt1 uses a loop. Loops are much more lightweight in terms of both heap and CPU usage due to the lack of overhead that a stream uses.
  2. From your experimentation, which method do you recommend using for each category when used to solve a competitive programming problem. Why?
     1. I recommend using findIt1() to solve a competitive programming problem because it’s more efficient in terms of both runtime and memory.

1. sumIt1, libSumIt
   1. sumIt1()

|  |  |  |
| --- | --- | --- |
| Elements | Runtime | Memory Usage |
| 10 | 1600 | 7 |
| 100 | 1100 | 7 |
| 1,000 | 9800 | 7 |
| 10,000 | 157,600 | 7 |
| 100,000 | 923,200 | 7 |

* 1. libSumIt()

|  |  |  |
| --- | --- | --- |
| Elements | Runtime | Memory Usage |
| 10 | 37,452,500 | 10 |
| 100 | 17,500 | 10 |
| 1,000 | 82,000 | 10 |
| 10,000 | 369,200 | 10 |
| 100,000 | 1,411,900 | 10 |

* 1. Research about each method. Why is one faster/slower more/less a memory hog than the others?
     1. The difference here is the same as the last category of methods. libSumIt uses an Array stream and sumIt1 uses a loop. Loops are much more lightweight in terms of both heap and CPU usage due to the lack of overhead that a stream uses.
  2. From your experimentation, which method do you recommend using for each category when used to solve a competitive programming problem. Why?
     1. I recommend using sumIt1() to solve a competitive programming problem because it’s more efficient in terms of both runtime and memory.

1. allSum1, allSum2
   1. allSum1()

|  |  |  |
| --- | --- | --- |
| Elements | Runtime | Memory Usage |
| 10 | 2700 | 7 |
| 100 | 56,800 | 7 |
| 1,000 | 2,040,400 | 7 |
| 10,000 | 21,711,400 | 7 |
| 100,000 | 1,447,479,600 | 10 |

* 1. allSum2()

|  |  |  |
| --- | --- | --- |
| Elements | Runtime | Memory Usage |
| 10 | 1900 | 7 |
| 100 | 1800 | 7 |
| 1,000 | 16,800 | 7 |
| 10,000 | 240,500 | 7 |
| 100,000 | 1,165,500 | 7 |

* 1. Research about each method. Why is one faster/slower more/less a memory hog than the others?
     1. allSum2 is faster, especially with higher input sizes, because its runtime complexity is only O(n) while allSum1’s runtime complexity is O(n2). There’s a memory usage difference with higher input sizes likely related to how much extra addition is done, which creates many new values in the heap and redirects the array positions to them.
  2. From your experimentation, which method do you recommend using for each category when used to solve a competitive programming problem. Why?
     1. I recommend using allSum2() to solve a competitive programming problem because it’s more efficient in terms of both runtime and memory, especially with higher input sizes.

1. sortIt1, sortIt2, libSortIt
   1. sortIt1()

|  |  |  |
| --- | --- | --- |
| Elements | Runtime | Memory Usage |
| 10 | 2800 | 7 |
| 100 | 63,000 | 7 |
| 1,000 | 2,160,200 | 7 |
| 10,000 | 49,508,300 | 7 |
| 100,000 | 2,752,704,400 | 10 |

* 1. sortIt2()

|  |  |  |
| --- | --- | --- |
| Elements | Runtime | Memory Usage |
| 10 | 13,500 | 7 |
| 100 | 19,300 | 7 |
| 1,000 | 50,500 | 7 |
| 10,000 | 333,600 | 7 |
| 100,000 | 1,198,400 | 7 |

* 1. libSortIt()

|  |  |  |
| --- | --- | --- |
| Elements | Runtime | Memory Usage |
| 10 | 209,300 | 7 |
| 100 | 16,900 | 7 |
| 1,000 | 300,100 | 7 |
| 10,000 | 1,881,200 | 7 |
| 100,000 | 5,162,200 | 7 |

1. Research about each method. Why is one faster/slower more/less a memory hog than the others?
   * 1. Like the previous category of methods, sortIt2 is faster than sortIt1 because sortIt2’s runtime complexity is O(n) while sortIt1’s is O(n2). libSortIt uses the Arrays.sort method which, according to Java 8 documentation, is a quicksort algorithm when using primitive data types. Quicksort’s runtime complexity is O(n\*log(n)), which explains why it’s almost as good as sortIt2, but not quite the same. Also, like the previous category of methods, there’s a memory usage difference with higher input sizes in sortIt1. This is likely related to the amount of time it takes to loop through the rest of the array from each position in it and swap values during most iterations.
2. From your experimentation, which method do you recommend using for each category when used to solve a competitive programming problem. Why?
3. I recommend using sortIt2() to solve a competitive programming problem because it’s the most efficient in terms of both runtime and memory, especially with higher input sizes.