# Worksheet #2

# CS 249 – Data Structures

23 September 2014

by

Brandon Horner

1. Determine runtime complexity of each of the following code segments (A) through (D) below. Count the number of “basic instructions” performed in the code as well as the number of times loops execute (in terms of the number of data elements N) and express the overall complexity using Big-O notation. (NOTE: show ALL of your work).

(A) This method to sums the values in an array:

public static long sum(int[] values){

long sum = 0; //1

for(int i=0; i<values.length; i++){ //(2, 3, 4) \* N

sum += values[i]; //5\*N

}

return sum; //6

}

// The loop will run N times, which means it could have to go through all of the array or only some of it. The complexity is O(N).

(B) A method that averages an array of values; it calls the method from (A):

public static long avg(int[] values){

long sum = sum(values); //1

long avg = sum/values.length; //2, 3

System.out.print("Average of: { "); //4

for(int i=0; i<values.length; i++) //(5, 6, 7)\*N

System.out.print(values[i] + " "); //8 \* N

System.out.print("} is " + avg + "\n"); //9

return avg; //10

}

// The loop will run N times, dependent on how big the array is. The big O efficiency of this method is O(N).

(C) An implementation of a search algorithm on a sorted array.

public int mySearch(int[] sorted,int first,int upto,int key){

while (first < upto) { //1

int mid = (first + upto) / 2; //2

if (key < sorted[mid]) //3 \* N

upto = mid; //4 \* N

else if (key > sorted[mid]) //3 \* N

first = mid + 1; //4 \* N

else //3 \* N

return mid; //4 \* N (only 1 sets of these

basic operations will run)

}

return -(first + 1); //5, 6

}

//This while loop will execute a said number of times, but worst case scenario still achieves O(N) efficiency (If the start were at the front of the array, and upto was the end).

(D) A method that uses the methods from question (A) and (B):

public void foo(int[] vals){

long sum = sum(vals); //1 \* N (uses O(N) method)

long avg = avg(vals); //2 \* N (^ ^ ^ )

long sum2 = sum(vals); //3 \* N (^ ^ ^ )

System.out.print("Values: { "); //4

for(int i=0; i<vals.length; i++) //(5, 6, 7) \* N

System.out.print(vals[i] + " "); //8 \* N

System.out.println("}"); //9

System.out.print("Values: { "); //10

for(int i=vals.length-1; i>=0; i--) //(11, 12, 13) \* N

System.out.print(vals[i] + " "); //14 \* N

System.out.println("}"); //15

}

//Each of these for loops will have to run N times for a worst case scenario. However they are not nested, so the combined efficiency of this method is just O(N).

1. “Draw” the structure and elements of an array being used to implement a stack as the following push and pop statements are executed: push(11), push(2), push(55), push(4), pop(), pop(),push(70), push(20), push(3), pop().Then display the contents of the stack from top to bottom.

|  |
| --- |
| 0 |
| 0 |
| 0 |
| 0  Top |
| 11 |

|  |
| --- |
| 0 |
| 0 |
| 0 |
| 11 |
| 2 |

|  |
| --- |
| 0 |
| 0 |
| 11 |
| 2 |
| 55 |

|  |
| --- |
| 0 |
| 11 |
| 2 |
| 55 |
| 4 |

|  |
| --- |
| 0 |
| 0 |
| 11 |
| 2 |
| 55 |

|  |
| --- |
| 0 |
| 11 |
| 2 |
| 70 |
| 20 |

|  |
| --- |
| 0 |
| 0 |
| 0 |
| 11 |
| 2 |

|  |
| --- |
| 0 |
| 11 |
| 2 |
| 70 |
| 20 |

|  |
| --- |
| 0 |
| 0 |
| 11 |
| 2 |
| 70 |

|  |
| --- |
| 11 |
| 2 |
| 70 |
| 20 |
| 3 |

Final

push (11) = {11}

push(2) = {2, 11}

push(55) = {55, 2, 11}

push(4) = {4, 55, 2, 11}

pop()= {55, 2, 11} // return 4

pop()= {2, 11} //return 55

push(70) = {70, 2, 11}

push(20) = {20, 70, 2, 11}

push(3) = {3, 20, 70, 2, 11}

pop()= **{20, 70, 2, 11}** //return 3, final contents, top to bottom.

1. “Draw” the structure and elements of an array being used to implement a queue (which wraps around) as well as the front and rear references, after each of the following numbers are inserted or removed from the queue. You may assume the queue can hold up to 5 elements. insert(11), insert(2), insert(55), insert(4), remove(), remove(), insert 70(), insert 20(), insert 3(), remove(). Then display the contents of the queue from front to rear.

insert(11) = {11, 0, 0, 0, 0} //front = 0 rear = 0 (index)

insert(2) = {11, 2, 0, 0, 0} //front = 0, rear = 1

insert(55) = {11, 2, 55, 0, 0} //front = 0, rear = 2

insert(4) = {11, 2, 55, 4, 0} //front = 0, rear = 3

remove() = {0, 2, 55, 4, 0} // front = 1, rear = 3

remove() = {0, 0, 55, 4, 0} //front = 2, rear = 3

insert(70) = {0, 0, 55, 4, 70} //front = 2, rear = 4

insert(20) = {20, 0, 55, 4, 70} //front = 2, rear = 0

insert(3) = {20, 3, 55, 4, 70} //front = 2, rear = 1

remove() = {20, 3, 0, 4, 70} //front = 3, rear = 1

**{4, 70, 20, 3}** //(queue contents from front to rear)