Ryder Dale Walton

Prof. David North

Data Structures and Algorithms

1/26/2016

Section 1: Each of these algorithms is evaluated for the total number of operations, as best as I am able to count them and consider what the machine is doing while it executes. These counts are then used to analyze the algorithms asymptotically.

**//R-4.9**

**public** **static** **int** example1(**int**[] arr) {

**int** n = arr.length, total = 0; //3

**for**(**int** j = 0; j < n; j++){ //2n +2

total += arr[j]; //2n

}

**return** total; //1

}

R-4.9 has a single loop that performs n iterations in it. The first line in the method contains a length operation, and two assignment operations, which adds to 3. The for loop head contains a single initialization for 1, a conditional expression that executes n+1 times (since it has to perform an additional check and determine that the expression is false in order to exit the for loop), and the increment occurs n times. This sums to 2n+2. The inside of the loop happens the same number of times j is incremented, which is n. There are two operations in the body, so they add to 2n operations. Finally, the return statement is one operation. When these are all summed together, I estimate that there are 4n+6 total operations, which means this method is **O(n)**.

//R-4.10

**public** **static** **int** example2(**int**[] arr){

**int** n = arr.length, total = 0; //3

**for**(**int** j = 0; j < n; j += 2){ //(2n/2)+2

total += arr[j]; //(2n/2)

}

**return** total; //1

}

R-4.10 has a single for loop with n/2 iterations in it since j increments by multiples of 2. The general makeup of the array is the same as in part one. The incrementing is the only difference. See the comments on the right side of the code for my counts. Since the for loop executes half of the number of times, the n is divided by two. When I sum up all of the counts I have above, I get 2n+6. This method is **O(n)** as well.

//R-4.11

**public** **static** **int** example3(**int**[] arr){

**int** n = arr.length, total = 0; //3

**for**(**int** j = 0; j < n; j++){ //2n+2

**for**(**int** k = 0; k <= j; k++){ //[(n(n+1))/2] + 2n

total += arr[j]; //[(n(n+1))/2]\*2

}

}

**return** total; //1

}

R-4.11 has nested for loops, which tends to mean we will be operating in O(n2) time. I have provided my counts above again. The nested for loop executes varying numbers of times based on the number of times the outer loop executes. The innards of the outer loop should execute n times because this is the number of times the loop increments. That means that the head of the inner loop should execute 1 + 2 + 3 + … + n times, which equals [(n(n+1))/2]. Additionally, the inner loop initializes a variable and performs an additional check to exit the for loop 2n times when they are summed together. The innards of the nested loop execute the number of times the counter increments, which is also [(n(n+1))/2]. Each time the innards execute, 2 operations are performed, so that value must be multiplied by two. Everything else remains the same. When these are added together, I get something in the ballpark of (3/2)n2 + (11/2)n +6 operations. This is **O(n2).**

//R-4.12

**public** **static** **int** example4(**int**[] arr){

**int** n = arr.length, prefix = 0, total = 0; //4

**for**(**int** j = 0; j < n; j++){ //2n+2

prefix += arr[j]; //2n

total += prefix; //n

}

**return** total; //1

}

R-4.12 returns to the single for loop design, having only O(1) operations scattered through it. My counts are above. I count 5n+7 operations, which is **O(n)** time.

//R-4.13

**public** **static** **int** example1(**int**[] first, **int**[] second){

**int** n = first.length, count = 0; //3

**for**(**int** i = 0; i < n; i++){ //2n+2

**int** total = 0; //n

**for**(**int** j = 0; j < n; j++){ //n(2n+2)

**for**(**int** k = 0; k <= j; k++){ //[(n(n+1))/2]\*n

total += first[k]; //[(n(n+1))/2]2n

}

}

**if**(second[i] == total){ //2n

count++; //0-n

}

}

**return** count; //1

}

R-4.13 is by far the messiest algorithm to count. I have listed my counts above as I did for the other four examples. The first line is an access and two assignments, so that is 3. The head of the first for loop executes n times, giving us 2n+2 operations. The first line in the body of the outer four loop is a simple assignment that executes n times. The second four loop also executes 2n+2 times in and of itself. However, since it is in the body of the first four loop and runs all the way to n, it must be multiplied by n to account for every time it runs. The same is true for the innermost for loop except that it runs to j, rather than to n. In terms of n, this translates to the familiar [(n(n+1))/2]. However, because it runs inside both of the outer for loops, it needs to be multiplied by an additional n. The body of the innermost for loop has an operation that performs 2 operations times the number of for loop runs it performs: [(n(n+1))/2]\*n. The if statement is nested within only the outermost for loop, so the condition is checked n times, and an array is accessed n times, for a total of 2n. If the condition never clears, the count will never increment and run 0 times, but we should always account for the worst case scenario. In this instance, that is that the count would increment once every time the outermost for loop executed, which would sum to n times. Finally, the return is one operation. If I did my math correctly, this would sum to (3/2)n3 + (7/2)n2 + 8n + 6. That would make this function be **O(n3)**.

//**C-4.36**

The following is an algorithm used to collect the ten largest elements of an array into a smaller array. The order in the smaller array was not specified, and my only constraint is that the algorithm be efficient.

Algorithm: big10()  
Input: Array *A* of size *n*.  
Output: Array *B* of size 10 that holds the largest elements.

1. A.mergeSort() //mergeSort the large array
2. bIndex=0;
3. **For** k=(n-1) to k=(n-10) **do** //always iterates 10 times
   1. B[i]= A[k]; //two different indices are kept since
   2. i++; //arrays are counting from different ends
4. Return B;

In terms of speed, this algorithm relies on a merge sort, which is O(n log n). The incrementing through the array is nothing O(n) because it is nothing but assignment, access, and comparison operations. Therefore, the running time of the algorithm is **O(n log n)**. Since the loop executes ten times every call, this algorithms formula would be something like n log n + 64.

//**C-4.36**

Since I must find the wine in only one month, and the scheme must be O(log n) for n bottles, I must use a divide and conquer approach to try and triangulate the bottle. First, I will take a simple example of there being 8 bottles. To triangulate the bottle, I need three servants to drink from 4 bottles apiece in different patterns. The first servant drinks from the first four bottles sequentially, following this pattern: xxxxoooo, where an x is a bottle from which the servant drank. The second servant alternates through the bottles two at a time: xxooxxoo. The third servant alternates every bottle: xoxoxoxo. Now, depending on which of the servants dies, we may know which bottle contained the poison since each of the 8 bottles results in a different combination of the servants dying or living. For example, if the first bottle was poisoned (the left x in all three servants’ rows), then all three servants died. This is the only possibility in which all servants die. At the end of the month, therefore, the king and his guards will know which bottle to dispose of.

Note that 23 is 8. If there are eight bottles, then it takes three servants to drink all of them. Log2 8 is three, so we have an O(log n) scheme for pinpointing the poison bottle. Simply add more servants based on this formula and take alternating paths as described above, and that is the solution for n bottles.

**//P-4.61**

**EXAMPLE ONE:**

**MAIN:**

**public** **static** **void** main(String[] args) **throws** IOException{

**int** range = 100000;

**int** increment = 100;

**long**[] results = **new** **long**[range/increment];

**int**[] nItems = **new** **int**[range];

**long** start = 0;

**long** stop = 0;

**int** total = 0;

File file = **new** File("assn4Ex1\_1.txt");

FileWriter fileWriter = **new** FileWriter(file);

BufferedWriter bufferedWriter = **new** BufferedWriter(fileWriter);

PrintWriter printWriter = **new** PrintWriter(bufferedWriter);

**int** j = 0;

**for**(**int** i = 1; i < range; i += increment){

start = System.*nanoTime*();

total = *example1*(nItems, i);

stop = System.*nanoTime*();

nItems[j] = i;

results[j] = stop-start;

System.***out***.println(nItems[j] + ", " + results[j]);

printWriter.println(nItems[j] + "," + results[j]);

j++;

}

printWriter.close();

}

**TESTED ALGORITHM:**

//5-4.9

**public** **static** **int** example1(**int**[] arr, **int** limit) {

**int** n = limit, total = 0;

**for**(**int** j = 0; j < n; j++){

total += arr[j];

}

**return** total;

}

**GRAPH:**

Observe the trendline. It is linear, telling us the algorithm is O(n). There are several fluctuations in performance time on my machine, but it is clear that the trend is linear.

**EXAMPLE 2**

**MAIN:**

**public** **static** **void** main(String[] args) **throws** IOException{

**int** range = 100000;

**int** increment = 100;

**long**[] results = **new** **long**[range/increment];

**int**[] nItems = **new** **int**[range];

**long** start = 0;

**long** stop = 0;

**int** total = 0;

File file = **new** File("assn4Ex2\_1.txt");

FileWriter fileWriter = **new** FileWriter(file);

BufferedWriter bufferedWriter = **new** BufferedWriter(fileWriter);

PrintWriter printWriter = **new** PrintWriter(bufferedWriter);

**int** j = 0;

**for**(**int** i = 1; i < range; i += increment){

start = System.*nanoTime*();

total = *example2*(nItems, i);

stop = System.*nanoTime*();

nItems[j] = i;

results[j] = stop-start;

System.***out***.println(nItems[j] + ", " + results[j]);

printWriter.println(nItems[j] + "," + results[j]);

j++;

}

printWriter.close();

}

**ALGORITHM TESTED:**

//5-4.10

**public** **static** **int** example2(**int**[] arr, **int** limit){

**int** n = limit, total = 0;

**for**(**int** j = 0; j < n; j += 2){

total += arr[j];

}

**return** total;

}

**CHART:**

Again, the trendline is linear, which is what I expected based on my analysis from the first portion of this homework. I admit that my machine fluctuated wildly on this one, but it is clear that the trend is linear.

**EXAMPLE 3**

**MAIN:**

**public** **static** **void** main(String[] args) **throws** IOException{

**int** range = 100000;

**int** increment = 100;

**long**[] results = **new** **long**[range/increment];

**int**[] nItems = **new** **int**[range];

**long** start = 0;

**long** stop = 0;

**int** total = 0;

File file = **new** File("assn4Ex3\_1.txt");

FileWriter fileWriter = **new** FileWriter(file);

BufferedWriter bufferedWriter = **new** BufferedWriter(fileWriter);

PrintWriter printWriter = **new** PrintWriter(bufferedWriter);

**int** j = 0;

**for**(**int** i = 1; i < range; i += increment){

start = System.*nanoTime*();

total = *example3*(nItems, i);

stop = System.*nanoTime*();

nItems[j] = i;

results[j] = stop-start;

System.***out***.println(nItems[j] + ", " + results[j]);

printWriter.println(nItems[j] + "," + results[j]);

j++;

}

printWriter.close();

}

**ALGORITHM TESTED:**

//5-4.11

**public** **static** **int** example3(**int**[] arr, **int** limit){

**int** n = limit, total = 0;

**for**(**int** j = 0; j < n; j++){

**for**(**int** k = 0; k <= j; k++){

total += arr[j];

}

}

**return** total;

}

**CHART:**

This one makes me happy. It’s almost perfect! The trendline is buried in the points. It is O(n2).

**EXAMPLE 4:**

**MAIN:**

**public** **static** **void** main(String[] args) **throws** IOException{

**int** range = 100000;

**int** increment = 100;

**long**[] results = **new** **long**[range/increment];

**int**[] nItems = **new** **int**[range];

**long** start = 0;

**long** stop = 0;

**int** total = 0;

File file = **new** File("assn4Ex4\_1.txt");

FileWriter fileWriter = **new** FileWriter(file);

BufferedWriter bufferedWriter = **new** BufferedWriter(fileWriter);

PrintWriter printWriter = **new** PrintWriter(bufferedWriter);

**int** j = 0;

**for**(**int** i = 1; i < range; i += increment){

start = System.*nanoTime*();

total = *example4*(nItems, i);

stop = System.*nanoTime*();

nItems[j] = i;

results[j] = stop-start;

System.***out***.println(nItems[j] + ", " + results[j]);

printWriter.println(nItems[j] + "," + results[j]);

j++;

}

printWriter.close();

}

**ALGORITHM TESTED:**

//5-4.12

**public** **static** **int** example4(**int**[] arr, **int** limit){

**int** n = limit, prefix = 0, total = 0;

**for**(**int** j = 0; j < n; j++){

prefix += arr[j];

total += prefix;

}

**return** total;

}

**CHART:**

This is another linear O(n) method, which matches up with my prediction in the first section of this homework.

**EXAMPLE 5:**

**MAIN:**

**public** **static** **void** main(String[] args) **throws** IOException{

**int** range = 1000;

**int** increment = 10;

**long**[] results = **new** **long**[range/increment];

**int**[] nItems = **new** **int**[range];

**long** start = 0;

**long** stop = 0;

**int** total = 0;

File file = **new** File("assn4Ex5\_1.txt");

FileWriter fileWriter = **new** FileWriter(file);

BufferedWriter bufferedWriter = **new** BufferedWriter(fileWriter);

PrintWriter printWriter = **new** PrintWriter(bufferedWriter);

**int** j = 0;

**for**(**int** i = 1; i < range; i += increment){

start = System.*nanoTime*();

total = *example5*(nItems, nItems, i);

stop = System.*nanoTime*();

nItems[j] = i;

results[j] = stop-start;

System.***out***.println(nItems[j] + ", " + results[j]);

printWriter.println(nItems[j] + "," + results[j]);

j++;

}

printWriter.close();

}

**ALGORITHM TESTED:**

//5-4.13

**public** **static** **int** example5(**int**[] first, **int**[] second, **int** limit){

**int** n = limit, count = 0;

**for**(**int** i = 0; i < n; i++){

**int** total = 0;

**for**(**int** j = 0; j < n; j++){

**for**(**int** k = 0; k <= j; k++){

total += first[k];

}

}

**if**(second[i] == total){

count++;

}

}

**return** count;

}

**CHART:**

As predicted above, example5 functions in O(n3) time. The trendline supports my conclusion.