CSCI E-119
Section Notes
Section 3

Big O notation

Some basic concepts.

1.) constant multipliers do not affect Big-O
    `10000n`^{2} `and 0.00005n`^{2} `are both O(n`^{2}`)`
2.) lower order terms do not affect Big-O
    `2`^{n} `+ n`^{1000} `is still O(2`^{n}`)`
3.) General Ordering: O(1) < O(`log n) < O(n) < O(n log n) < O(n`^{k}`)< O(c`^{n})

Time Analysis Example: Standard Matrix Multiplication

The formula for matrix multiplication is:

`[a_ij]*[b_ij] = [c_ij]`

`c_ij = Sum_{k = 1 to N} (a_ik * b_kj)`

`If we write this using for loops:`

```
for (int i = 0; i < n; i++){
      for (int j = 0; j < n; j++){
            double total = 0.0;
            for (int k = 0; k < n; k++){
                  total += a[i][k] * b[k][j]
            }
      }
}
```
What is the number of multiplications needed by standard matrix multiplication?
What is the complexity of this method in terms of the number of multiplications performed?

To obtain each of the n^2 entries in the matrix c, we perform n multiplications. Thus, the total number of multiplications performed is n^3. This algorithm is O(n^3).

More generally, if you have nested loops, and the outer loop performs O(m) iterations and the inner loop performs O(n) iterations, the number of operations performed by the code inside the inner loop is O(m * n). In the case of matrix multiplication, we have three nested loops, each of which performs n = O(n) iterations, and thus the number of operations performed is O(n * n * n) or O(n^3).

We can also apply this rule of thumb for nested loops to selection sort. In that case, the number of operations performed by the outer loop is n – 1 = O(n), and the number of operations performed by the inner loop (the loop that finds the smallest remaining element) ranges from n – 1 down to 1, but that is still O(n) on average. Thus, we can conclude that selection sort performs O(n * n) = O(n^2) operations. While it is possible to perform a more formal mathematical derivation of the number of operations performed, as we did in lecture, this more informal type of analysis is often sufficient.

Tips on Analyzing the Time Complexity of Iterative Algorithms

Here are several examples of how to analyze the time complexity of iterative algorithms that employ nested loops.

If you have nested loops, and the outer loop iterates i times and the inner loop iterates j times, the statements inside the inner loop will be executed a total of i * j times. This is because the inner loop will iterate j times for EACH of the i iterations of the outer loop.

This means that if *both* the outer and inner loop are dependent on the problem size n, the statements in the inner loop will be executed O(n^2) times:

```
for ( int i = 0; i < n; ++i ) {
    for ( int j = 0; j < n; ++j ) {
        // these statements are executed O(n^2) times
     }
}
```

Likewise, if you have triply-nested loops, all of which are dependent on the problem size n, the statements in the innermost loop will be executed O(n^3) times:

```
for ( int i = 0; i < n; ++i ) {
    for ( int j = 0; j < n; ++j ) {
        for ( int k = 0; k < n; ++k ) {
            // these statements are executed O(n^3) times
        }
     }
}
```

However, imagine a case with doubly-nested loops where *only* the outer loop is dependent on the problem size n, and the inner loop always executes a *constant* number of times, say 3 times:

```
for ( int i = 0; i < n; ++i ) {
    for ( int j = 0; j < 3; ++j ) {
        // these statements are executed O(n) times
    }
}
```

In this particular case, the inner loop will execute exactly 3 times for each of the n iterations of the outer loop, and so the total number of times the statements in the innermost loop will be executed is 3n or O(n) times, NOT O(n^2) times.
Now, imagine a third case: you have doubly nested loops, and the outer loop is dependent on the problem size n, but the inner loop is dependent on the current value of the index variable of the outer loop:

```
for ( int i = 0; i < n; ++i ) {
    for ( int j = 0; j < i; ++j ) {
        // these statements are executed O(n^2) times
    }
}
```

Let's analyze this case iteration-by-iteration:
On the 1st iteration of the outer loop (i = 0), the inner loop will iterate 0 times
On the 2nd iteration of the outer loop (i = 1), the inner loop will iterate 1 time
On the 3rd iteration of the outer loop (i = 2), the inner loop will iterate 2 times
.
.
On the *final* iteration of the outer loop (i = n - 1), the inner loop will iterate n - 1 times
So, the *total* number of times the statements in the inner loop will be executed will be equal to the sum of the integers from 1 to n - 1, which is ((n - 1)*n)/2 = n^2/2 - n/2 = O(n^2) times

Selection Sort

```
-------------------------------
|  7 | 39 | 20 | 11 | 16 |  5 |
-------------------------------
-------------------------------
|  5 | 39 | 20 | 11 | 16 |  7 |
-------------------------------
-------------------------------
|  5 |  7 | 20 | 11 | 16 | 39 |
-------------------------------
-------------------------------
|  5 |  7 | 11 | 20 | 16 | 39 |
-------------------------------
-------------------------------
|  5 |  7 | 11 | 16 | 20 | 39 |
-------------------------------
-------------------------------
|  5 |  7 | 11 | 16 | 20 | 39 |
-------------------------------
```

```
private static int indexSmallest(int[] arr, int lower, int upper){
    int indexMin = lower;
      for (int i = lower+1; i <= upper; i++)
        if (arr[i] < arr[indexMin])
          indexMin = i;
      return indexMin;
}
public static void selectionSort(int[] arr) {
    for (int i = 0; i < arr.length-1; i++) {
      int j = indexSmallest(arr, i, arr.length-1);
      swap(arr, i, j);
    }
```

What is the big-O time complexity of selection sort in the best case? The worst case? The average case?

Selection sort always performs O(n2) comparisons and O(n) moves.So, it is O(n2) overall in the best, worst, and average cases.

Insertion Sort

```
-------------------------------
|  7 | 39 | 20 | 11 | 16 |  5 |
-------------------------------
-------------------------------
|  7 | 39 | 20 | 11 | 16 |  5 |
-------------------------------
-------------------------------
|  7 | 20 | 39 | 11 | 16 |  5 |
-------------------------------
-------------------------------
|  7 | 11 | 20 | 39 | 16 |  5 |
-------------------------------
-------------------------------
|  7 | 11 | 16 | 20 | 39 |  5 |
-------------------------------
-------------------------------
|  5 |  7 | 11 | 16 | 20 | 39 |
-------------------------------
```

```
public static void insertionSort(int[] arr) {
  for (int i = 1; i < arr.length; i++) {
    if (arr[i] < arr[i-1]) {
      int toInsert = arr[i];
      int j = i;
      do {
        arr[j] = arr[j-1];
        j = j - 1;
      } while (j > 0 && toInsert < arr[j-1]);
      arr[j] = toInsert;
    }
  }
}
```

What is the big-O time complexity of the insertion sort in the best case? The worst case? The average case?
Best Case: O(n) (for almost-sorted data) Worst Case: O(n2) Average Case:O(n2)

Bubble Sort

```
-------------------------------
|  7 | 39 | 20 | 11 | 16 |  5 |
-------------------------------
-------------------------------
|  7 |  20 | 11 | 16 | 5 | 39 |
-------------------------------
-------------------------------
|  7 |  11 | 16 | 5 | 20 | 39 |
-------------------------------
-------------------------------
|  7 |  11 | 5 | 16 | 20 | 39 |
-------------------------------
-------------------------------
|  7 |  5 | 11 | 16 | 20 | 39 |
-------------------------------
-------------------------------
|  5 |  7 | 11 | 16 | 20 | 39 |
-------------------------------
```

```
public static void bubbleSort(int[] arr) {
  for (int i = arr.length – 1; i > 0; i--) {
    for (int j = 0; j < i; j++) {
      if (arr[j] > arr[j+1])
        swap(arr, j, j+1);
    }
  }
}
```

What is the big-O time complexity of bubble sort in the best case? The worst case? The average case?

The best, worst, and average cases for bubble sort are all O(n2)

Tracing through Shell Sort

Let's trace what happens when the supplied array is processed by the shellSort() method below.

```
public static void shellSort(int[] arr) {
   int incr = 1;
   while (2 * incr <= arr.length)
      incr = 2 * incr;
   incr--;

   while (incr >= 1) {
      for (int i = incr; i < arr.length; i++) {
         if (arr[i] < arr[i - incr]) {
            int toInsert = arr[i];

            int j = i;
            while (j > 0 && toInsert < arr[j - incr]) {
               arr[j] = arr[j - incr];
               j = j - incr;
            }

            arr[j] = toInsert;
         }
      }

      incr = incr / 2;
   }
}
```

```
   0    1    2    3    4    5       increment    index                      move/no move
-------------------------------
|  7 | 39 | 20 | 11 | 16 |  5 |     incr = 3     i = 3
-------------------------------
-------------------------------
|  7 | 39 | 20 | 11 | 16 |  5 |     incr = 3     i = 3     toInsert = 11     ( nothing )
-------------------------------
-------------------------------
|  7 | 16 | 20 | 11 | 39 |  5 |     incr = 3     i = 4     toInsert = 16     ( 16 < 39 )
-------------------------------
-------------------------------
|  7 | 16 |  5 | 11 | 39 | 20 |     incr = 3     i = 5     toInsert = 5      ( 5  < 20 )
-------------------------------
-------------------------------
|  7 | 16 |  5 | 11 | 39 | 20 |     incr = 1     i = 1     toInsert = 16     ( nothing )
-------------------------------
-------------------------------
|  5 |  7 | 16 | 11 | 39 | 20 |     incr = 1     i = 2     toInsert = 5      (5 < 7, 16)
-------------------------------
-------------------------------
|  5 |  7 | 11 | 16 | 39 | 20 |     incr = 1     i = 3     toInsert = 11     ( 11 < 16 )
-------------------------------
-------------------------------
|  5 |  7 | 11 | 16 | 39 | 20 |     incr = 1     i = 4     toInsert = 39     ( nothing )
-------------------------------
-------------------------------
|  5 |  7 | 11 | 16 | 20 | 39 |     incr = 1     i = 5     toInsert = 20     ( 20 < 39 )
-------------------------------
```