Plan for 241 day 3:DETERMINING THE RUNNING TIME OF ITERATIVE ALGORITHMS USING SUMMATIONS.

We need a way to determine the time taken (3 methods):

a) Experimental method: code the algorithm and time the execution on a given computer platform. This solution suffers from the fact that processors may be under different loads and that one processor might be slower than another (too machine dependent).

b) Count the time taken by every line of execution and sum the running time for each on data size n, assuming all simple operations take constant time (yields messy and lengthy polynomial expressions).

c) Identify the operation(s) that are executed most often (basic or dominant operation). In an iterative algorithm, this is generally the operation(s) executed in the innermost loop of nested loops. Much easier to determine the count of one line than it is for entire algorithm. Assume each line takes a constant amount of time.

ITERATIVE ALGORITHMS:

Process for determining running time of iterative algorithms:

1. Decide on a parameter indicating input size.
2. Identify the algorithm's basic operation.
3. Set up a sum expressing the number of times the basic operation is executed in the worst case.

   If number of times basic operation is executed depends on something other than data size n, give a sum for the best-case number of times too.
4. Find a function or functions describing the sum of operations.
5. Express function(s) found in step 4 asymptotically (using O or Theta).

```
MaxElement(A[1..n])
  Input:  A is an array of real numbers and x is a number.
  Output: The largest element in A
                                          cost    times
      maxval = A[0]                        c1       1
      for i = 2 to n                       c2       n

        if A[i] > maxval                   c3       n-1
          maxval = A[i]                    c4       n-1

      return maxval                        c5       1
```

Ask: What is basic operation?
Different running times on different input instances?
Are there b-c and w-c running times? If so, give instances of each.
What is running time in asymptotic notation?

```
MatrixMultiplication(A[1..n],B[1..n])
  Input:  2 n-by-n matrices
  Output: Matrix C = AB

     for i = 1 to n
        for j = 1 to n
            C[i,j] = 0.0
            for k = 1 to n
                 C[i][j] = C[i,j] + A[i,k] * B[k,j]
     return C
```

Ask: What is basic operation?
Different running times on different input instances?
Are there b-c and w-c running times? If so, give instances of each.
What is running time in asymptotic notation?

```
UniqueElements(A[1..n])
  Input:  A is an array [1..n] of comparable items
  Output: true if all elts are distinct and false o.w.
                                          cost    times
     for i = 1 to n                        c1       ??

       for j = i+1 to n                    c2       ??

          if A[i] = A[j]                   c3       ??
            return false                   c4       ??

     return true                           c5       1
```

Ask: What is basic operation?
Different running times on different input instances?
Are there b-c and w-c running times? If so, give instances of each.
What is running time in asymptotic notation?

```
LinearSearch(x, A):
  Input:   A is an array of whole numbers and x is a number.
  Purpose: Return index of number x in array A or -1 if x does not
  appear in A.
                                          cost    times
    i = 1                                 c1       1
    while (i <= A.length)                 c2       ??

        if (A[i] == x)                    c3       ??
            return i                      c4       ??

    return -1                             c5        1
```

Does this algorithm have same running time for all valid inputs? No

Ask: What is basic operation?
Different running times on different input instances?
Are there b-c and w-c running times? If so, give instances of each.
What is running time in asymptotic notation?

Omega(1) is best case.

```
O(n) is worst case.
Theta(n/2) is average case.
```

```
Algorithm for sorting problem:

   InsertionSort(A):
      Input:  A is an array of whole numbers <a1 a2 a3 ... an>.
      Output: Permutation of input array <a1' a2' a3' ... an'> such that
      a1' <= a2' <= a3' <= ... <= an'
                                          cost    times
       for j = 2 to n                      c1       n

           key = A[j]                      c2      n-1
           i = j - 1                       c3      n-1
           while (i > 0 and A[i] > key)    c4       ti

               A[i+1] = A[i]               c5       ti - 1  ??
               i = i - 1                   c6       ti - 1  ??

           A[i+1] = key                    c7      n-1
```

Does this algorithm have same running time for all valid inputs? No

Ask: What is basic operation? Different running times on different input instances? Are there b-c and w-c running times? If so, give instances. What is running time in asymptotic notation?

$\Omega(n)$ is best case. What does the data set look like for this running time?
$\mathcal{O}(n^2)$ is worst case. What does the data set look like for this running time?
$\Theta(\frac{n}{2})$ is average case.

Notes:
- Every start of a for loop is checked one more time than any operation

inside the loop.
- The basic operation is the while loop, executed
$\sum_{j=2}^{n} j = 2 + 3 + ... + n = \sum_{j=1}^{n} j - 1$ in the worst case.

```
FACTs ABOUT SUMMATIONS:

      sum (lower to upper) 1 = upper - lower + 1

      sum(i=1 to n-1) sum(j=0 to i-1)1, where sum(j=0 to i-1)1 = i-1 - 0 + 1 = i =
      sum(i=1 to n-1)i = (n-1)n/2

      sum (i=1 to n) i = n(n+1)/2, so sum (i=1 to n-1) i = (n-1)n/2.
```

NEXT SUBJECT: Divide-and-conquer algorithms (Chapter 4)

Binary Search

Merge-Sort (ch. 1)