Algorithm Analysis

Running Time of a Program

1. We would like an algorithm that is easy to understand, code, and debug.
2. We would like an algorithm that makes efficient use of the computer's resources, especially, one that runs as fast as possible.

Choosing an Algorithm

1. Simplicity
2. Clarity
3. Efficiency
4. The time taken to run the program
5. The amount of storage space taken by its variables
6. Amount of traffic it generates on a network of computers

Choosing an Algorithm…

- The amount of data that must be moved to and from disks.
  - When a program is to be run repeatedly, running time of the program is a major factor in choosing one algorithm over another.

Measuring the Running Time

- The running time of a program depends on factors such as:
  - The input to the program
  - The quality of code generated by the compiler used to create the object program
  - The nature and speed of the instructions on the machine used to execute the program
  - The time complexity of the algorithm underlying the program

Measuring Running Time…

- Once we have agreed that we can evaluate a program by measuring its running time, we face the problem of determining what the running time actually is.
  - Benchmarking.
    - Profiler.
    - Statement counter (90- 10 principle).
  - Analysis.

Running Time …

- It is customary to talk of $T(n)$, to be the running time of a program on inputs of size n.
- The units of $T(n)$ will be left unspecified, but we can think of $T(n)$ as being the number of instructions executed on an idealized computer.

Big-oh Notation

- We say that $T(n)$ is $O(f(n))$ if there are *positive* constants c and n_0 such that $T(n) \leq c\, f(n)$ whenever $n \geq n_0$. A program whose running time is $O(f(n))$ is said to have growth rate of $f(n)$

Big-Omega notation

- Just as *O*-notation provides an asymptotic upper bound on a function, Ω notation provides an asymptotic lower bound. $T(n)$ is said to be $\Omega(g(n))$, if there exist *positive* constants c and n_0 such that

  $$T(n) \geq c.g(n) \text{ for all } n \geq n_0.$$

Θ - notation

- $T(n)$ is said to be $\Theta(g(n))$, if there exist *positive* constants $c1$ and $c2$ and n_0 such that

$$c_1.g(n) \le T(n) \le c_2.g(n) \text{ for all } n \ge n_0.$$

- Θ is the most desirable bound to have because it provides a realistic time complexity for our algorithm.

Example 1

```
public void mystery(int n) {
    int i, j, k;
      for (i = 1; i <= n; i++)
         for (j = 1; j <= n; j++)
            for (k = 1; k <= n - 1; k++)
               {some statement requiring constant time}
}
```

Example 2

```
public void mystery(int n) {
    int i, j, k;
      for (i = 1; i <= n - 1; i++)
          for (j = i +1; j <= n; j++)
             for (k = 1; k <= j; k++)
             {some statement requiring constant time}
}
```

Example 3

```
public void veryodd (int n )
    int x = 0, y  = 0;
        for (int i = 1; i <= n;  i++)
            if (i % 2 != 0) {  // i is odd
                for (int j = i; j <= n; j++)
                    x = x + 1;
                for (int j = 1; j <= i; j++)
                    y = y + 1;
            }
}
```

Example 4

```
public void  mystery (int n ) {
   int x, count;
     count = 0;
      x = 1;
      while (x < n) {
            x  = 2 * x;
            count  = count + 1;
      }
    System.out.println (count);
}
```

Example 5

```
public void matmpy(int  n ) {
    int i, j, k;
        for (i = 0;  i < n; i++)
            for (j = 0; j < n; j++) {
                C[i] [j] = 0;
                for (k = 0; k < n; k++)
                    C[i] [j] = C[i] [j] + A[i] [k] * B[k] [j];
            }
}
```

Example 6

```
public void mystery (int data[], int count) {
    int minIndex, temp;
        for (int ii = 0; ii < count - 1; ii++)  {
            minIndex = ii;
            for (int jj = ii+1; jj < count ; jj++)
                if ( data[jj] < data[minIndex])
                    minIndex = jj;
    // swap the elements at positions minIndex and ii
            temp = data[minIndex];
            data[minIndex] = data[ii];
            data[ii] = temp;
        }
}
```

Example 7

```
// data is a sorted array of integers;  count is the # of  elements in the array
public int binarySearch(int data[], int count, int target) {
   int mid;
   int first = 0, last = count - 1;
   while (first <= last) {
         mid = (first + last)/2;
        if (data[mid] == target)    // we are done.  Successful search
                return mid;
        if (target < data[mid])
                 // target has to be in the first half of the array
                last = mid - 1;
        else  // target has to be in the latter half of the array
                first = mid + 1;
   }  // while
   return -1;  // unsuccessful search
  }
```

Example 8

```
public int recursive(int n) {
   if (n <= 1) return (1);
   else return(recursive(n-1) + recursive(n-1));
}
```