CSCI 112
Thursday, July 15, 2010

Analysis of Algorithms

There are two metrics for measuring the efficiency of an algorithm.

- **Time**
- **Space**

The way to record time in java is you record the beginning system time as a *long* variable, then do the same with the system time at the end. The difference of the two is the length of time it took to run.

Example Code: Timing An Algorithm

```
long start = System.currentTimeMillis();
//some operations
System.out.println( "Runtime: " + ( System.currentTimeMillis() - start ) + " milliseconds");
```

Because all computers have different speeds, this isn't an accurate benchmark between systems.

Big O Notation

Study of the number of basic operations performed by an algorithm

Primitive Operations:

- basic assignment: x=1;
- calling methods
- arithmetic
- comparison of two numbers
- indexing into an array
- dereferencing objects
- return from a method

Seven Basic Growth Functions (from fastest to slowest):

- **$O(1)$ – Constant Time**
- **$O(\log_2 N)$ – Logarithmic Time**
- **$O(N)$ – Linear Time**
- **$O(N*\log_2 N)$ – Log-Linear Time**
- **$O(N^2)$ – Square Time**
- **$O(N^3)$ – Cubed Time**
- **$O(2^n)$ – Exponential Time**

Constant Time Algorithms - O(1)

```
int sum( int x ) {
        return ( x * ( x + 1 ) ) / 2;  // 3
}
```

This total basic operations for this code is: 3

Our Big O Notation shows this to be in **O(1)** time. This algorithm has **constant** time.

Logarithmic Algorithms - O($\log_2$N)

This is the second fastest type of algorithm we can have. If you have a range of 100 elements the most choices you have before finding what you want (assuming they are in order) is $\log_2 100$ which is ~8 choices.

Linear Algorithms - O(N)

```
int sum( int n ) {
        int sum = 0;  // 1
        for( int i = 1; i <= n; i++ ){ // 1, n, n, 2n
                sum+=i;
        }
}
```

The total basic operations of this code is: 4n+2

So our Big O Notation shows this to be in **O(N)** time. This algorithm has l**inear** time.

Log-Linear Algorithms - O(N*$\log_2$N)

You don't see these very often, but two very famous algorithms that use this are:

- quicksort
- mergesort

Square Time Algorithms - O(N^2)

For every element in a list, touch every other element. This means if you have 100 elements, you do 10000 operations. Most sorting algorithms fit into this class.

- insertion sort
- selection sort
- bubble sort

Cubed Time – O(N^3)

- Matrix Multiplication

Exponential Time – O(2^n)

Won't be seen very often, not commercialized, as they can only be used for very specific problems that can't be solved any other way.

The Clock Game (The Number Guessing Game)

The Clock Game is logarithmic time

On the game show, “The Price is Right,” Bob Barker played a game called, “The Clock Game.” In this game, Bob told the contestant that the value of “the item up for bid” was between a high and low value. The user then had to guess the value of the item within 30 seconds. After each guess, Bob would say “higher” if the guess was too low or “lower” if the guess was too high. It is the only game on “The Price is Right” where it is possible to win every time without knowing anything about the product as long as the contestant follows a logarithmic time algorithm.

Psuedocode

```
low <- Bob's Low Bound
high <- Bob's High Bound
secret <- Bob's Secret Value
while (clock is ticking)
   guess <- (high + low) / 2
   if (secret < guess)
      Bob Says "LOWER!"
      high <- guess + 1
   end if
   if (secret > guess)
      Bob Says "HIGHER!"
      low <- guess + 1
   end if
   if (secret > guess)
      Bob Says "RIGHT!"
      break
   end if
end while
```

Binary Search

This code is a binary search algorithm that essentially plays the clock game.

```
int BinarySearch( int[] nums, int value ){
        int low = 0;
        int high = nums.length - 1;

        while( low <= high){
                int guess = ( high - low ) / 2 + low;
                if( value < nums[guess] ){
                        high = guess - 1;
                }
                else if( value > nums[guess] ){
                        low = guess + 1;
                }
                if( value == nums[guess] ){
                        return guess;
                }
        }
        return -1;
}
```

Basic Sorting

Bubble Sort

- Group every pair of side by side numbers into overlapping bubbles.
- For each bubble, if it is out of order, swap the values.
- Repeat until all bubbles no longer require swapping.

```
void BubbleSort(int nums){
        for( int i = 0; i < nums.length - 1; i++ ){
                boolean swapped = false;

                for( int j = 0; j < nums.length - 1; j++ ){
                        if( nums[j] > nums[j+1] ){
                                int temp = nums[j];
                                nums[j] = nums[j+1];
                                nums[j+1] = temp;
                                swapped = true;
                        }
                }
                if( swapped == false ){
                        break;
                }
        }
}
```