Algorithm Analysis

Complexity Analysis

- Establishing the relationship between the input size and the algorithm/ program time (and/or space) requirement.
  - Estimate the time and space requirement for a given input size
  - Compare algorithms
- Time Complexity: counting operations
  - Count the number of operations that an algorithm will perform
- Asymptomatic analysis
  - The Big O notation
  - How fast time/space requirements increase as the input size approaches infinity

```
// number of outputs
// t(n) = n
for(i = 0; i < n, i++){
	cout << A[i] << endl;
}

// number of comparisons
// t(n) = n-1
template<class T>
bool IsSorted(T *A, int n){
	bool sorted = true;
	for(int i=0; i<n-1; i++){
		if(A[i]) > A[i+1]){
			sorted = false;
		}
	}
	return sorted;
}
```

Algorithm analysis covers the worst case (most of the time).
The average case is more useful, however, it is more difficult to calculate.
The complexity of a function is its input size. For instance...
$t(n) = 1000n$ vs. $t(n) = 2n^2$
if n doubled...
$t(n) = 1000n...1000 * 2n/1000n = 2$

- Time Will Double
  $t(n) = 2n^2 ... 2(2n^2) = 4n^3$
- Time increases by 4x

Scaling Analysis

- The constant factor does not change the growth rate and can be ignored
- **We can ignore the slower-growing terms.**
  - Ex. $n^2 + n + 1$ ... n^2
  - capturing $O(n^2)$

Example Problem

Algorithm 1: $t_1(n) = 100n + n^2$

- insert - n, delete - $log(n)$, lookup - 1
  Algorithm 2: $t_2(n) = 10n^2$
- insert - $log(n)$, delete - n, lookup - $log(n)$
  Which is faster if an application has as many inserts but few deletes and lookups?

Asymptotic Complexity Analysis

- Compares the growth rate of two functions
- Variables & Values - Nonnegative integers
- Dependent on eventual (asymptotic) behavior
- Independent of constant multipliers, and lower-order effects

Big "O" Notation

$f(n) = O(g(n))$
$iff \exists c, n_0 > 0 | 0 < f(n) < cg(n) \forall n >= n_0$

- if there exists two positive constants $c > 0$ & $n_0 > 0$ such that $f(n) \leq cgn(n)$ for all $n \geq n_0$

Big "Omega" Notation

$f(n) = \Omega(g(n))$
$iff \exists c, n_0 > 0 | 0 < cg(n) < f(n) \forall n >= n_0$

- $f(n)$ is asymptotically lower bounded by $g(n)$

Big "Theta" Notation

$f(n) = \theta(g(n))$
$iff \exists c_1, c_2, n_0 > 0 | 0 < c_1 g(n) < f(n) < c_2 g(n) \forall n >= n_0$

- $f(n)$ has the same long-term growth as $g(n)$

Examples

$f(n) = 3n^2 + 17$

$\Omega(1), \Omega(n), \Omega(n^2)$ -> lower bounds

- Chose $\Omega(n^2)$ because it is the closest to the lower bound
- Set $\exists$ to 3. So... $3(n^2)$
  $O(n^2), O(n^3)$ -> upper bounds
- Chose $O(n^3)$ because it is the closest to the upper-bound
  $\theta(n^2)$ -> exact bound
  Why f(n) != O(n)?

Transitivity

$f(n) = O(g(n))$ -> $(a <= b)$
$f(n) = \Omega(g(n))$ -> $(a >= b)$
$f(n) = \theta(g(n))$ -> $(a = b)$

Additive property

- If $e(n) = O(g(n))$ and $f(n)$ = $O(h(n))$
- Then...
  - $e(n) + f(n) = O(g(n)) + O(h(n))$

Function	Name
c	Constant
$log(N)$	Logarithmic
$log^2(N)$	Log-squared
N	Linear
$NlogN$	
N^2	Quadratic
N^3	Cubic
2^n	Exponential

Running time Calculations - Loops

```
for(j=0; j < n; ++j){
// 3 Atomics
}
```

- Each iteration has 3 atomics so $3n$

- Cost of the iteration itself ($c * n$, c is a constant)
- Complexity $\theta(3n + c * n) = \theta(n)$

Running time Calculations - Loops with a break

```
for(j=0; j < n; ++j){
// 3 Atomics
        if(condition) break;
}
```

- Upper bound - $O(4n) = (n)$
- Lower bound - $\Omega(4) = \Omega(1)$
- Complexity - $O(n)$

Complexity Analysis

- Find n = input size
- Find atomic activities count
- Find $f(n)$ = the number of atomic activities done by an input

Sequential Search

```
for(size_t i= 0; i < a.size() ;i++){
        if(a[i] == x){return}
}
// θ(n) time complexity
```

If then for loop

```
if(condition) i=0;
else
        for(j =0; j < n; j++)
                a[j] = j;
// θ(n) time complexity
```

Nested Loop Search

```
for(j =0; j < n; j++){
        // 2 atomics
        for(k =0; k<n; k++){
                // 3 atomics
        }

```

```
}
//θ(n^2) time complexity
```