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

// number of comparisons
// t(n) = n-1
template
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 <strong>worst
case</strong> (most of the time).
The average case is more useful, however, it is more
```

```
difficult to calculate.
The complexity of a function is its <strong>input
size</strong>. 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

<h3>Scaling Analysis</h3>
- The constant factor does not change the growth rate
and can be ignored
- <strong>We can ignore the slower-growing terms.
</strong>
        - Ex. $n^2+n+1$ ... $n^2$
        - capturing $O(n^2)$

<h3>Example Problem</h3>
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?

<h3>Asymptotic Complexity Analysis</h3>
- Compares the <strong>growth rate</strong> of two
functions
```

```
- Variables & Values - Nonnegative integers
- Dependent on eventual (asymptotic) behavior
- Independent of constant multipliers, and lower-order effects

<h3>Big "O" Notation</h3>
$f(n)=O(g(n))$
$iff ∃ c, n_0 > 0 | 0 < f(n) < cg(n) ∀ n >= n_0$
- if there exists two positive constants $c>0$ & $n_0>0$ such that $f(n) ≤ cgn(n)$ for all $n ≥ n_0$
<h3>Big "Omega" Notation</h3>
$f(n)=Ω(g(n))$
$iff ∃ c, n_0 > 0 | 0 < cg(n) < f(n) ∀ n >= n_0$
- $f(n)$ is asymptotically lower bounded by $g(n)$
<h3>Big "Theta" Notation</h3>
$f(n)=θ(g(n))$
$iff ∃ c_1, c_2, n_0 > 0 | 0 < c_1g(n) < f(n) < c_2g(n) ∀ n >= n_0$
- $f(n)$ has the same long-term growth as $g(n)$
<h3>Examples</h3>
$f(n) = 3n^2 + 17$

$Ω(1), Ω(n), Ω(n^2)$ -> <strong>lower bounds</strong>
- Chose $Ω(n^2)$ because it is the <strong>closest</strong> to the lower bound
- Set $∃$ to 3. So... $3(n^2)$
$O(n^2), O(n^3)$ -> <strong>upper bounds</strong>
- Chose $O(n^3)$ because it is the <strong>closest</strong> to the upper-bound
```

````
$θ(n^2)$ -> <strong>exact bound</strong>
<strong>Why f(n) != O(n)?</strong>

<h3>Transitivity</h3>
$f(n) = O(g(n))$ -> $(a <= b)$
$f(n) = Ω(g(n))$ -> $(a >= b)$
$f(n) = θ(g(n))$ -> $(a = b)$
<h3>Additive property</h3>
- 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
$N log N$|
$N^2$|Quadratic
$N^3$|Cubic
$2^n$|Exponential

<h3>Running time Calculations - <strong>Loops</strong>
</h3>
```cpp
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

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

If then for loop

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

Nested Loop Search

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