Analysis of Algorithms

CSE 373 - Data Structures
April 10, 2002

Asymptotic Behavior

- The “asymptotic” performance as $N \to \infty$, regardless of what happens for small input sizes $N$, is generally most important
- Performance for small input sizes may matter in practice, if you are sure that small $N$ will be common forever
- We will compare algorithms based on how they scale for large values of $N$

Big-Oh Notation

- The growth rate of the time or space required in relation to the size of the input $N$ is generally the critical issue
- $T(N)$ is said to be $O(f(N))$ if
  - there are positive constants $c$ and $n_0$ such that $T(N) \leq cf(N)$ for $N \geq n_0$.
  - i.e., $f(N)$ is an upper bound on $T(N)$ for $N \geq n_0$
- $T(N)$ is “big-oh” of $f(N)$ or "order" $f(N)$

Readings and References

- Reading
  - Chapter 2, *Data Structures and Algorithm Analysis in C*, Weiss

- Other References
Big-Oh Notation

- Suppose $T(N) = 50N$
  - $T(N) = O(N)$
  - Take $c = 50$ and $n_0 = 1$

- Suppose $T(N) = 50N+11$
  - $T(N) = O(N)$
  - $T(N) \leq 50N+11N = 61N$ for $N \geq 1$. So, $c = 61$ and $n_0 = 1$ works

The common comparisons

<table>
<thead>
<tr>
<th>Name</th>
<th>Big-Oh</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constant</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Log log</td>
<td>$O(\log \log N)$</td>
</tr>
<tr>
<td>Logarithmic</td>
<td>$O(\log N)$</td>
</tr>
<tr>
<td>Log squared</td>
<td>$O((\log N)^2)$</td>
</tr>
<tr>
<td>Linear</td>
<td>$O(N)$</td>
</tr>
<tr>
<td>$N \log N$</td>
<td>$O(N \log N)$</td>
</tr>
<tr>
<td>Quadratic</td>
<td>$O(N^2)$</td>
</tr>
<tr>
<td>Cubic</td>
<td>$O(N^3)$</td>
</tr>
<tr>
<td>Exponential</td>
<td>$O(2^N)$</td>
</tr>
</tbody>
</table>

Exponential Growth swamps everything else
Bounds

- Upper bound (O) is not the only bound of interest
- Big-Oh (O), Little-Oh (o), Omega (Ω), and Theta (Θ):
  - (Fraternities of functions…)
  - Examples of time and space efficiency analysis

Big-Oh Notation

Run Time

\[ T_A(N) = O(N^2) \]

\[ T_A(N) \text{ is } O(N^2) \text{ because } 50N \leq N^2 \text{ for } N \geq 50. \]

So \( N^2 \) is an upper bound. But it’s not a very tight upper bound.

Big-Oh and Omega

- \( T(N) = O(f(N)) \) if there are positive constants \( c \) and \( n_0 \) such that \( T(N) \leq cf(N) \) for \( N \geq n_0 \).
  - \( O(f(N)) \) is an upper bound for \( T(N) \)
  - 100 \( \log N \), \( N^{0.9} \), 0.0001 \( N \), \( 2^{100} N + \log N \) are \( O(N) \)
- \( T(N) = \Omega(f(N)) \) if there are positive constants \( c \) and \( n_0 \) such that \( T(N) \geq cf(N) \) for \( N \geq n_0 \).
  - \( \Omega(f(N)) \) is a lower bound for \( T(N) \)
  - \( 2^N \), \( N^{\log N} \), \( N^{1.2} \), 0.0001 \( N \), \( N + \log N \) are \( \Omega(N) \)

Theta and Little-Oh

- \( T(N) = \Theta(f(N)) \) iff \( T(N) = O(f(N)) \) and \( T(N) = \Omega(f(N)) \)
  - \( \Theta(f(N)) \) is a tight bound, upper and lower
  - 0.0001 \( N \), \( 2^{100} N + \log N \) are all = \( \Theta(N) \)
- \( T(N) = o(f(N)) \) iff \( T(N) = O(f(N)) \) and \( T(N) \neq \Theta(f(N)) \)
  - \( f(N) \) grows faster than \( T(N) \)
  - 100 \( \log N \), \( N^{0.9} \), \( \sqrt{N} \), 17 are all = \( o(N) \)
For large $N$ and ignoring constant factors

- $T(N) = O(f(N))$
  - means $T(N)$ is less than or equal to $f(N)$
  - Upper bound
- $T(N) = \Omega(f(N))$
  - means $T(N)$ is greater than or equal to $f(N)$
  - Lower bound
- $T(N) = \Theta(f(N))$
  - means $T(N)$ is equal to $f(N)$
  - “Tight” bound, same growth rate
- $T(N) = o(f(N))$
  - means $T(N)$ is strictly less than $f(N)$
  - Strict upper bound: $f(N)$ grows faster than $T(N)$

Big-Oh Analysis of iterative sum function

Find the sum of the first $\text{num}$ integers stored in array $v$. Assume $\text{num} \leq \text{size of } v$.

```java
int sum ( int v[ ], int num) {
    int temp_sum = 0, i; //1
    for ( i = 0; i < num; i++ ) //2
        temp_sum = temp_sum + v[i] ; //3
    return temp_sum; //4
}
```

- lines 1, 3, and 4 take fixed (constant) amount of time
- line 2: $i$ goes from 0 to $\text{num}-1=\text{num}$ iterations
- Running time = constant + $(\text{num})*\text{constant} = O(\text{num})$
- Actually, $\Theta(\text{num})$ because there are no fast cases

Big-Oh Analysis of recursive sum function

Recursive function to find the sum of first $\text{num}$ integers in $v$.

```java
int sum ( int v[ ], int num){
    if (num == 0) return 0; //constant time here
    else return v[num-1] + sum(v,num-1); // constant + $T(\text{num-1})$
}
```

- Let $T(\text{num})$ be the running time of $\text{sum}$
- Then, $T(\text{num}) = \text{constant} + T(\text{num}-1)$
- $= 2^*\text{constant} + T(\text{num}-2) = \ldots = \text{num}^*\text{constant} + \text{constant}$
- $= \Theta(\text{num})$ (same as iterative algorithm!)

Common Recurrence Relations

- Common recurrence relations in analysis of algorithms:
  - $T(N) = T(N-1) + \Theta(1) \Rightarrow T(N) = O(N)$
  - $T(N) = T(N-1) + \Theta(N) \Rightarrow T(N) = O(N^2)$
  - $T(N) = T(N/2) + \Theta(1) \Rightarrow T(N) = O(\log N)$
  - $T(N) = 2T(N/2) + \Theta(N) \Rightarrow T(N) = O(N \log N)$
  - $T(N) = 4T(N/2) + \Theta(N) \Rightarrow T(N) = O(N^2)$
Big-Oh Analysis of Recursive Algorithms

- To derive, expand the right side and count
- Note: Multiplicative constants matter in recurrence relations:
  - \( T(N) = 4T(N/2) + \Theta(N) \) is \( O(N^2) \), not \( O(N \log N) \)!
- You will see these again later
  - you will only need to know a few specific relations and their big-oh answers

Recursion

- Recall the example using Fibonacci numbers
  - 1, 1, 2, 3, 5, 8, 13, 21, 34, …
  - First two are defined to be 1
  - Rest are sum of preceding two
    - \( F_n = F_{n-1} + F_{n-2} \) (n > 1)

Recursive Fibonacci Function

```c
int fib(int N) {
    if (N < 0) return 0; //invalid input
    if (N == 0 || N == 1) return 1; //base cases
    else return fib(N-1) + fib(N-2);
}
```

- Running time \( T(N) = ? \)

Recursive Fibonacci Analysis

```c
int fib(int N) {
    if (N < 0) return 0; // time = 1 for (N < 0)
    if (N == 0 || N == 1) return 1; // time = 3
    else return fib(N-1)+fib(N-2); // T(N-1)+T(N-2)+1
}
```

- Running time \( T(N) = T(N-1) + T(N-2) + 5 \)
- Using \( F_n = F_{n-1} + F_{n-2} \) we can show by induction that \( T(N) \geq F_N \). We can also show by induction that \( F_N \geq (3/2)^N \)
- Therefore, \( T(N) \geq (3/2)^N \)
  - Exponential running time!
Iterative Fibonacci Function

```c
int fib_iter(int N) {
    int fib0 = 1, fib1 = 1, fibj = 1;
    if (N < 0) return 0; //invalid input
    for (int j = 2; j <= N; j++) { //all fib nos. up to N
        fibj = fib0 + fib1;
        fib0 = fib1;
        fib1 = fibj;
    }
    return fibj;
}
```

• Running time = ?

Iterative Fibonacci Analysis

```c
int fib_iter(int N) {
    int fib0 = 1, fib1 = 1, fibj = 1;
    if (N < 0) return 0; //invalid input
    for (int j = 2; j <= N; j++) { //all fib nos. up to N
        fibj = fib0 + fib1;
        fib0 = fib1;
        fib1 = fibj;
    }
    return fibj;
}
```

• Running time $T(N) = \text{constant} + (N-1) \cdot \text{constant}$
• $T(N) = \Theta(N)$
  » Exponentially faster than recursive Fibonacci

Appendix

Matlab - bigo.m

```matlab
% bigO functions
% calculate and plot several functions used in comparing growth rates
figure
n = 1:100; % the x axis
y1 = n-1; % O(1)
y2 = log2(n); % O(log2(n))
y3 = n; % O(n)
y4 = n.*log2(n); % O(nlog2(n))
y5 = n^2; % O(n^2)
y6 = 2^n; % O(2^n)
plot(n,y1,'y','LineWidth',2)
hold on
plot(n,y2,'r','LineWidth',2)plot(n,y3,'g','LineWidth',2)plot(n,y4,'b','LineWidth',2)plot(n,y5,'m','LineWidth',2)
plot(n,y6,'c','LineWidth',2)
legend('O(1)','O(log_2 n)','O(n)','O(n log_2 n)','O(n^2)','O(2^n)',2)
hold off
```