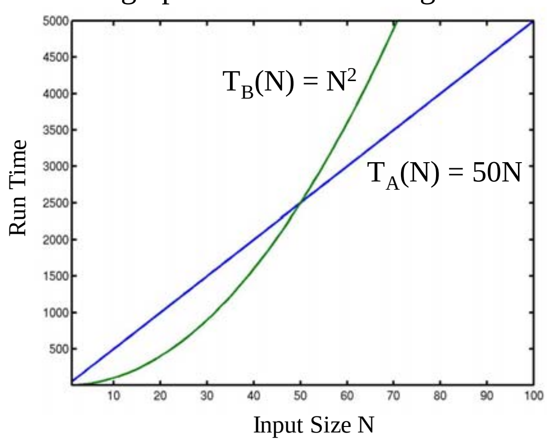
CSE 326 Lecture 3: Analysis of Algorithms

- Today, we will review:
  - Big-Oh, Little-Oh, Omega (Ω), and Theta (Θ): (Fraternities of functions…)
  - Examples of time and space efficiency analysis
- Covered in Chapter 2 of the text

Recall from Last Time: Big-Oh Notation

- The graph shows the running times of algorithms A and B:



$T_A(N) = O(T_B(N))$

$T_A(N)$ is "Big-Oh" of $T_B(N)$ because $50N \le N^2$ for $N \ge 50$.

Actually, for $N \ge 51$, $50N < N^2$

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 $N \geq n_0$.
- E.g. 100 log N, 53, $N^{0.99}$, 0.0001 N, 2^{100} N + log N are all = O(N)
- What if $T(N) \geq cf(N)$ for $N \geq n_0$?

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 $N \geq n_0$.
- E.g. 100 log N, 53, $N^{0.99}$, 0.0001 N, 2^{100} N + log N are all = O(N)
- $T(N) = \Omega(f(N))$ if there are positive constants c and n_0 such that $T(N) \geq cf(N)$ for $N \geq n_0$.
- E.g. 2^N, $N^{\log N}$, $N^{1.002}$, 0.0001 N, N + log N are all = $\Omega(N)$
- What if T(N) is both O(f(N)) and $\Omega(f(N))$?

Theta and Little-Oh

- $T(N) = \Theta(f(N))$ if and only if $T(N) = O(f(N))$ and $T(N) = \Omega(f(N))$
- E.g. $0.0001\ N$, $2^{100}\ N + \log N$ are all $= \Theta(N)$
- $T(N) = o(f(N))$ iff $T(N) = O(f(N))$ and $T(N) \neq \Theta(f(N))$
- E.g. $100 \log N$, $N^{0.9}$, $\text{sqrt}(N)$, 17 are all $= o(N)$

Big-Oh, Omega, Theta, and Little-Oh

- Tips to guide your intuition:
- Think of $O(f(N))$ as "less than or equal to" $f(N)$
  - Upper bound, "grows slower than or same rate as" $f(N)$
- Think of $\Omega(f(N))$ as "greater than or equal to" $f(N)$
  - Lower bound, "grows faster than or same rate as" $f(N)$
- Think of $\Theta(f(N))$ as "equal to" $f(N)$
  - "Tight" bound, same growth rate
- Think of $o(f(N))$ as "strictly less than" $f(N)$…
  - Strict upper bound
  - $T(N) = o(f(N))$ means $T(N)$ grows strictly slower than $f(N)$
- (*True for large N and ignoring constant factors*)

Big-Oh Analysis: Example 1

Problem: Find the sum of the first num integers stored in array v. Assume num ≤ size of v.

```
public static int sum ( int [ ] v, int num)
  {
  int temp_sum = 0;
  for (  int i = 0; i < num; i++  )
       temp_sum += v[i] ;
  return temp_sum;
  }
```

Running time = ?

Big-Oh Analysis: Example 1

Problem: Find the sum of the first num integers stored in array v. Assume num ≤ size of v.

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

- i goes from 0 to num-1= num iterations
- lines 1, 3, and 4 take fixed (constant) amount of time
- Running time = constant + (num)*constant = O(num)
- Actually, Θ(num)

Big-Oh Analysis: Example 1 (Recursion)

Recursive function to find the sum of the first num integers stored in array v:

```
public static int sum ( int [ ] v, int num)
  {
    if (num == 0) return 0;
     else return sum(v,num-1) + v[num-1];
  }
```

- Running time = ?

Big-Oh Analysis: Example 1 (Recursion)

Recursive function to find the sum of first num integers in v:

```
public static int sum ( int [ ] v, int num)
  {
    if (num == 0) return 0; // constant time T1 for "if"
     else return sum(v,num-1) + v[num-1];
          // constant time + T(num-1) = T2 + T(num-1)
  }
```

- Let T(num) be the running time of sum
- Then, $T(num) = T_1 + T_2 + T(num-1) = c + T(num-1)$
- $= 2*c + T(num-2) = \ldots = num*c + T(0) = num*c + c_1$
- $= \Theta(num)$ (same as iterative algorithm!)

Recurrence Relations for Run Time Analysis

- Common recurrence relations in analysis of algorithms:
  - $T(N) = T(N-1) + \Theta(1) \Rightarrow T(N) = O(N)$
  - $T(N) = T(N-1) + \Theta(N) \Rightarrow T(N) = O(N^2)$
  - $T(N) = T(N/2) + \Theta(1) \Rightarrow T(N) = O(\log N)$
  - $T(N) = 2T(N/2) + \Theta(N) \Rightarrow T(N) = O(N \log N)$
  - How do you get these? Just expand the right side and count!
- Note: Multiplicative constants matter in recurrence relations:
  - If $T(N) = 4T(N/2) + \Theta(N)$, then is $T(N) = O(N)$? $O(N \log N)$? $O(N^2)$?

Recurrence Relations for Run Time Analysis

- Common recurrence relations in analysis of algorithms:
  - $T(N) = T(N-1) + \Theta(1) \Rightarrow T(N) = O(N)$
  - $T(N) = T(N-1) + \Theta(N) \Rightarrow T(N) = O(N^2)$
  - $T(N) = T(N/2) + \Theta(1) \Rightarrow T(N) = O(\log N)$
  - $T(N) = 2T(N/2) + \Theta(N) \Rightarrow T(N) = O(N \log N)$
- Note: Multiplicative constants matter in recurrence relations:
  - $T(N) = 4T(N/2) + \Theta(N)$ is **$O(N^2)$,** not $O(N \log N)$!

These recurrences in their full glory in future lectures we will see…

Example 2: Fibonacci Numbers

- Recall our old friend Signor Fibonacci and his 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)

Leonardo Pisano Fibonacci (1170-1250)

Example 2: Recursive Fibonacci

```
public static 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) = ?

Example 2: Recursive Fibonacci

```
public static int fib(int N) {
  if (N < 0) return 0;  // time = 1 for the < operation
  if (N == 0 || N == 1) return 1;  // time = 3 for 2 ==, 1 ||
  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 $F_N \geq (3/2)^N$

Example 2: Recursive Fibonacci

```
public static int fib(int N) {
  if (N < 0) return 0;  // time = 1 for the < operation
  if (N == 0 || N == 1) return 1;  // time = 3 for 2 ==, 1 ||
  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, $T(N) \geq (3/2)^N$ i.e. $T(N) = \Omega((1.5)^N)$

Yikes…exponential running time!

Example 2: Iterative Fibonacci

```
public static 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 = ?

Example 2: Iterative Fibonacci

```
public static int fib_iter(int N) {
      int fib0 = 1, fib1 = 1, fibj = 1; // constant time
      if (N < 0) return 0;  // constant time
      for (int j = 2; j <= N; j++) { //N-1 iterations
            fibj = fib0 + fib1; // constant time
            fib0 = fib1; // constant time
            fib1 = fibj; // constant time
      }
      return fibj;   }
```

- Running time =
  $T(N) = \text{constant}+(N-1)\bullet\text{constant} = \Theta(N)$

Example 2: Iterative Fibonacci

- public static int fib_iter(int N) {

```
int fib0 = 1, fib1 = 1, fibj = 1; // constant time
if (N < 0) return 0;  // constant time
for (int j = 2; j <= N; j++) { //N-1 iterations
        fibj = fib0 + fib1; // constant time
        fib0 = fib1; // constant time
        fib1 = fibj; // constant time
}
return fibj;    }
```

- Running time =
  T(N) = constant+(N-1)•constant = Θ(N)
  ➪ Exponentially faster than recursive

Example 3: Time and Space Tradeoffs

- Problem DUP: Given an array A of n positive integers, are there any duplicates?
- For example, A: 34, 9, 40, 87, 223, 109, 58, 9, 71, 8
- An easy algorithm for DUP:

```
for (i = 0; i < N-1; i++)
    for (j = i+1; j < N; j++)
        if (A[i] == A[j]) {
            <print "Duplicates!">  return 0;}
<print "No Duplicates">
```

- Space required = ?

Example 3: Time and Space Tradeoffs

- Problem DUP: Given an array A of n positive integers, are there any duplicates?
- An easy algorithm for DUP:

```
for (i = 0; i < N-1; i++)
    for (j = i+1; j < N; j++)
        if (A[i] == A[j]) {
            <print "Duplicates!">  return 0; }
<print "No Duplicates">
```

- Space required (array + 2 variables) = N + 2 = Θ(N)
  - *Does not depend on size of values stored in A*
- Running time: How many steps in the worst case?

Example 3: Time and Space Tradeoffs

- Analyze the running time of easy algorithm for DUP:

```
for (i = 0; i < N-1; i++) // N-1 iterations
    for (j = i+1; j < N; j++) // N-i-1 iterations
        if (A[i] == A[j]) { // constant time c
            <print "Duplicates!"> return 0; }
<print "No Duplicates">
```

- Worst case = no duplicates. Total time = ?

$$\sum_{i=1}^{N-1}\sum_{j=i+1}^{N} c = \sum_{i=1}^{N-1} c(N-i-1) = c\sum_{i=1}^{N-1} N - c\sum_{i=1}^{N-1} i - c(N-1)$$

$$= cN(N-1) - c\frac{(N-1)N}{2} - c(N-1) = \Theta(N^2)$$

Example 3: Trading more space for less time

- New Algorithm for DUP:
  - Idea: Use A[i] as index into new array B initialized to 0's

```
for (i = 0; i < N; i++)
  if (B[A[i]] == 1) { // value in A[i] already present
    <print "Duplicates!">  return 0; }
  else B[A[i]] = 1;   // mark value in A[i] as present
<print "No Duplicates">
```

  - Similar to detecting collisions in hashing (Chapter 5)
- Worst Case Running Time = ?
- Space Required = ?

Example 3: Trading more space for less time

- New Algorithm for DUP:

```
for (i = 0; i < N; i++)
  if (B[A[i]] == 1) { // value in A[i] already present
    <print "Duplicates!">  return 0; }
  else B[A[i]] = 1;   // mark value in A[i] as present
<print "No Duplicates">
```

- Worst Case Running Time = $O(N)$
- Space Required = $O(2^m)$ where m is the number of bits required to represent the largest value that can potentially occur in A. E.g. $m = 8$ if max value of $A[i] = 255$.
- Prev. algorithm: more time [$\Theta(N^2)$] but less space [$\Theta(N)$]
- Such tradeoffs between space and time are common…

Example 4: Searching for an Item

- Problem: Search for an item X in a sorted array A. Return index of item if found, otherwise return –1.
- Brainstorming: What is an efficient way of doing this?

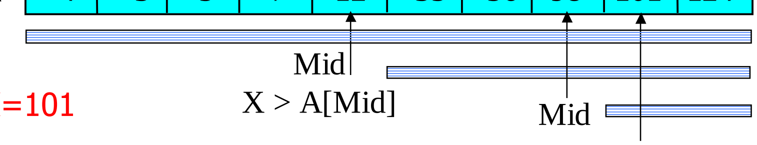
A										
	-4	-3	5	7	12	35	56	98	101	124

X=101

Example 4: Searching for an Item

- Problem: Search for an item X in a sorted array A. Return index of item if found, otherwise return –1.
- Idea: Compare X with middle item A[mid], go to left half if X < A[mid] and right half if X > A[mid]. Repeat.

A										
	-4	-3	5	7	12	35	56	98	101	124



X=101

A[Mid]=X
Found!
Return Mid = 8

Example 4: Binary Search

A	-4	-1	5	7	12	35	56	98	101	124

```
public static int BinarySearch( int [ ] A, int X, int N )
{
  int Low = 0, Mid, High = N - 1;
  while( Low <= High )    {
        Mid = ( Low + High ) / 2; // Find middle of array
        if ( X > A[ Mid ] )       // Search second half of array
            Low = Mid + 1;
        else if ( X < A[ Mid ] )          // Search first half
            High = Mid - 1;
        else return Mid;          // Found X!
        }
        return NOT_FOUND;
}
```


Example 4: Running Time of Binary Search

- Given an array A with N elements, what is the worst case running time of BinarySearch?
- Think about it over the weekend…
- We will discuss the answer in the next class

May the Big-Oh be with you…

To Do List:

Homework no. 1 (due Monday 11pm)

Begin reading Chapters 3 and 4

Next Week:

1. Review of Lists, Stacks, and Queues
2. The wonderful world of Trees!