

## CSE 143

### Program Efficiency

[Chapter 9, pp. 390-401]

2/20/99 224

## Does it matter?

- Faster is better
- Two approaches to improving performance
  - Better algorithm
  - Tighter code
- Other considerations
  - Development time
  - Ease of maintenance
  - Extensibility

2/20/99 225

## Fast tricky code

- C/C++ language/culture encourages tricky coding

```
while (*q++ = *p++) ;
```

- Reasons for caution
  - Correctness
  - Code used by others
  - No need to do compilers job
  - 90/10 principle

2/20/99 226

## Don't try this at home

- Copy count characters from from to to:

```
for (int i = 0; i < count; i++)  
    to[i] = from[i];
```

- Duff's way

```
n = (count + 3) / 4;  
switch (count % 4){  
case 0: do { *to++ = *from++;  
case 3:      *to++ = *from++;  
case 2:      *to++ = *from++;  
case 1:      *to++ = *from++;  
            } while (--n > 0);  
}
```

2/20/99 227

## Measuring Efficiency

- Ways of measuring efficiency
  - Run the program and see how long it takes
  - Run the program and see how much memory it uses
- Lots of variables to control
  - What input data?
  - What hardware platform?
  - What compiler? What compiler options?
- Just because one program is faster than another right now, will it always be faster?

2/20/99 228

## Complexity Analysis

- Lots of little details that we'll avoid, to achieve platform-independence
- Use an abstract machine that uses *steps* of time and *units* of memory, instead of seconds or bytes
  - Each elementary operation takes 1 step
  - Each elementary instance occupies 1 unit of memory
- Measure time and space in terms of the *size* of the input rather than details of the specific input
- Allows us to focus on big issues

2/20/99 229

## A Simple Example

```
// Input: int A[N], array of N integers
// Output: Sum of all numbers in array A
```

```
int Sum(int A[], int N) {
    int sum = 0;
    for ( int j = 0; j < N; j++ )
        sum = sum + A[j];
    return sum;
}
```

How should we analyze this?

2/20/99 230

## Analysis of Sum

- First, describe the *size* of the input in terms of one or more parameters
  - Input to `sum` is an array of  $N$  ints, so size is  $N$ .
- Then, count how many steps are used for an input of that size
  - A step is an elementary operation such as `+` or `<` or `A[j]`

2/20/99 231

## Analysis of Sum (2)

```
int Sum(int A[], int N) {
    int sum = 0; ①
    for ( int j = 0; j < N; j++ )
        sum = sum + A[j]; ② ③ ④ ⑤
    return sum; ⑥ ⑦ ⑧
}
```

- 1, 2, 8: Once
- 3, 4, 5, 6, 7: Once per each iteration of `for`-loop  $N$  iterations
- Total is  $5N + 3$  operations
- We can view this as a function of  $N$ , the **complexity function** of the algorithm:  $f(N) = 5N + 3$ .

2/20/99 232

## How $5N+3$ Grows

- The  $5N+3$  analysis gives an estimate of the true running time for different values of  $N$ :
  - $N = 10 \Rightarrow 53$  steps
  - $N = 100 \Rightarrow 503$  steps
  - $N = 1,000 \Rightarrow 5,003$  steps
  - $N = 1,000,000 \Rightarrow 5,000,003$  steps
- As  $N$  grows, the number of steps grows in *linear* proportion to  $N$ , for this `Sum` function

2/20/99 233

## What Dominates?

- What about the 5 in  $5N+3$ ? What about the  $+3$ ?
  - As  $N$  gets large, the  $+3$  becomes insignificant
  - The 5 is inaccurate:
    - `<`, `[]`, `+`, `=`, `++` require varying amounts of time
  - What is fundamental is that the time is *linear* in  $N$
- Asymptotic Complexity: As  $N$  gets large, concentrate on the *highest order* term
  - Drop lower order terms such as  $+3$
  - Drop the constant coefficient of the highest order term

2/20/99 234

## Asymptotic Complexity

- The  $5N+3$  time bound is said to “grow asymptotically” like  $N$
- This gives us an approximation of the complexity of the algorithm
- Ignores lots of details, concentrates on the bigger picture

2/20/99 235

## Comparing Algorithms

- We can now (partially) answer the question, "Given algorithms A and B, which is more efficient?"
- Same as asking which algorithm has the smaller asymptotic time bound
- For specific values of  $N$ , we might get different (and uninformative) answers
- Instead, compare the growth rates for arbitrarily large values of  $N$  (the *asymptotic* case)

2/20/99 236

## Comparing Functions

Definition: If  $f(N)$  and  $g(N)$  are two complexity functions, we say

$$f(N) = O(g(N))$$

(read " $f(N)$  is order  $g(N)$ ", or " $f(N)$  is big-O of  $g(N)$ ")

if there is a constant  $c$  such that

$$f(N) \leq c g(N)$$

for all sufficiently large  $N$ .

2/20/99 237

## Big-O Notation

- Think of  $f(N) = O(g(N))$  as " $f(N)$  grows at most like  $g(N)$ " or " $f$  grows no faster than  $g$ " (ignoring constant factors, and for large  $N$ )
- Big-O is not a function!
- Never read  $=$  as "equals"!
- Examples:
  - $5N + 3 = O(N)$
  - $37N^5 + 7N^2 - 2N + 1 = O(N^5)$
  - But also:  $5N + 3 = O(N^2)$

2/20/99 238

## Common Orders of Growth

Let  $N$  be the input size, and  $b$  and  $k$  be constants

$O(k) = O(1)$	Constant Time	Increasing Complexity ↓
$O(\log_b N) = O(\log N)$	Logarithmic Time	
$O(N)$	Linear Time	
$O(N \log N)$		
$O(N^2)$	Quadratic Time	
$O(N^3)$	Cubic Time	
$\dots$		
$O(k^N)$	Exponential Time	

2/20/99 239

## Why is this Useful?

What happens when we double the input size  $N$ ?

$N$	$\log_2 N$	$5N$	$N \log_2 N$	$N^2$	$2^N$
8	3	40	24	64	256
16	4	80	64	256	65536
32	5	160	160	1024	$\sim 10^9$
64	6	320	384	4096	$\sim 10^{19}$
128	7	640	896	16384	$\sim 10^{38}$
256	8	1280	2048	65536	$\sim 10^{76}$

2/20/99 240

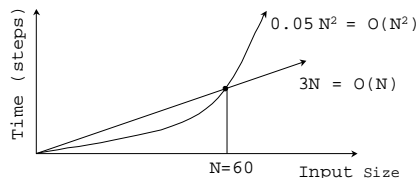
## Big numbers

- Suppose a program has run time  $O(n!)$  and the run time for  $n = 10$  is 1 second
- For  $n = 12$ , the run time is 2 minutes
- For  $n = 14$ , the run time is 6 hours
- For  $n = 16$ , the run time is 2 months
- For  $n = 18$ , the run time is 50 years
- For  $n = 20$ , the run time is 200 centuries

2/20/99 241

## Why is this Useful? (2)

- As inputs get larger, *any* algorithm of a smaller order will be more efficient than an algorithm of a larger order



2/20/99 242

## Big-O Arithmetic

- Remember common functions in order from smallest to largest:  
 $1, \log N, N, N \log N, N^2, N^3, \dots, 2^N$
- Ignore constant multipliers  
 $300 N + 5N^4 + 6 \cdot 2^N = O(N + N^4 + 2^N)$
- Ignore everything except the highest order term  
 $N + N^4 + 2^N = O(2^N)$

2/20/99 243

## Constant Time Statements

Simplest case:  $O(1)$  time statements

- Assignment statements of simple data types  
`int x = y;`
- Arithmetic operations  
`x = 5 * y + 4 * z;`
- Array referencing  
`A[j]`
- Referencing/dereferencing pointers  
`Cursor = Head -> Next;`
- Declarations of simple data types  
`int x, y;`
- Most conditional tests  
`if ( x < 12 ) ...`

2/20/99 244

## Constant Time Statements (2)

Watch out for things that look like simple  $O(1)$  time operations, but are actually more complex:

- Overloaded operators  
`LinkedList L1 (L2);` // deep copy?  
`Set s1 = s2 + s3;` // + for set union?
- Declaring complex data types that have constructors
- Function invocations  
`if ( anIntList.Size() < 10 ) ...`

2/20/99 245

## Analyzing Loops

Any loop analysis has two parts:

- How many iterations are performed?
- How many steps per iteration?

```
int sum = 0;
for (int j = 0; j < N; j++)
    sum = sum + j;
```

- Loop executes  $N$  times ( $0 \dots N-1$ )
- $4 = O(1)$  steps per iteration
- Total time is  $N \cdot O(1) = O(N \cdot 1) = O(N)$

2/20/99 246

## Analyzing Loops (2)

- What about this `for`-loop?

```
int sum = 0;
for (int j = 0; j < 100; j++)
    sum = sum + j;
```

- Loop executes 100 times ( $0 \dots 99$ )
- $4 = O(1)$  steps per iteration
- Total time is  $100 \cdot O(1) = O(100 \cdot 1) = O(100) = O(1)$
- That this loop is faster makes sense when  $N \gg 100$ .

2/20/99 247

## Analyzing Loops (3)

What about while-loops?

- Determine how many times the loop will be executed

```
bool done = false;
int result = 1, n;
cin >> n;
while ( !done ) {
    result = result * n;
    n--;
    if ( n <= 1 ) done = true;
}
```

- Loop terminates when `done == true`, which happens after `n` iterations
- $O(1)$  time per iteration
- $O(n)$  total time

2/20/99 248

## Nested Loops

- Treat just like a single loop, and evaluate each level of nesting as needed:

```
int j, k, sum = 0;
for ( j = 0; j < N; j++ )
    for ( k = N; k > 0; k-- )
        sum = sum + k + j;
```

- Start with outer loop:
  - How many iterations?  $N$
  - How much time per iteration? Need to evaluate inner loop ...
- Inner loop uses  $O(N)$  time
- Total is  $N \cdot O(N) = O(N \cdot N) = O(N^2)$

2/20/99 249

## Nested Loops (2)

- What if the number of iterations of one loop depends on the counter of the other?

```
int j, k, sum = 0;
for ( j = 0; j < N; j++ )
    for ( k = 0; k < j; k++ )
        sum += k * j;
```

- Analyze inner and outer loops together
- Number of iterations of the inner loop is  $0 + 1 + 2 + \dots + (N-1) = O(N^2)$
- Time per iteration is  $O(1)$ , for total  $O(N^2)$

2/20/99 250

## Sequences of Statements

For a sequence of statements, compute their cost functions individually and add them up

```
for ( int j = 0; j < N; j++ )
    for ( int k = 0; k < j; k++ )
        sum = sum + j*k;
for ( int l = 0; l < N; l++ )
    sum = sum - l;
cout << "Sum is now " << sum << endl; }
O(N^2)
O(N)
O(1)
```

Total cost is  $O(N^2) + O(N) + O(1) = O(N^2)$

2/20/99 251

## Conditional Statements

- What about a conditional statement such as

```
if (condition) statement1;
else statement2;
```

where `statement1` runs in  $O(n)$  time and `statement2` runs in  $O(n^2)$  time?

- We use "worst-case complexity": among all inputs of size `n`, what is the *maximum* running time?
- The analysis for the example above is  $O(n^2)$ .

2/20/99 252

## Recursive Algorithms

- We need to know two things:

- number of recursive calls
- the work done at each level of recursion

- Example: exponentiation

```
int exp (int x, int n) {
    if (n==0)
        return 1;
    else
        return x * exp(x,n-1);
}
```

- The running time is  $O(n)$ , because there are  $N$  recursive calls, and the work done at each call is constant

2/20/99 253

## Recursive Algorithms (2)

- Fibonacci numbers:

```
int fib (int n) {
    if (n == 1 || n == 2)
        return 1;
    else
        return fib(n-1) + fib(n-2);
}
```

- How many calls?
- How much work at each call?
- Running time?
- In general, a "recurrence relation" results from the analysis, solved with tools from math.

2/20/99 254

## List Implementations

N is the list size

	array	linked list	doubly linked list
constructor	O(1)	O(1)	O(1)
isEmpty	O(1)	O(1)	O(1)
isFull	O(1)	O(1)	O(1)
reset	O(1)	O(1)	O(1)
advance	O(1)	O(1)	O(1)
endOfList	O(1)	O(1)	O(1)
data	O(1)	O(1)	O(1)
size	O(1)	O(N)	O(N)
insertBefore	O(N)	O(N)	O(1)
insertAfter	O(N)	O(1)	O(1)
deleteItem	O(N)	O(N)	O(1)

2/20/99 255

## Where efficiency matters

- Passing parameters

- Pass by value -- copies entire structure
 

```
ostream & operator<<(ostream & s, IntList L);
```
- Pass by reference -- does not copy, but allows updates
 

```
ostream & operator<<(ostream & s, IntList & L);
```
- const reference -- pass by reference, but do not allow changes
 

```
ostream & operator<<(ostream & s, const IntList & L);
```

2/20/99 256

## Dynamic arrays

- When array is full, reallocate new array

- increase by one
 

```
if (size == maxSize){
    int *tmp = new int[maxSize + 1];
    ...
}
```
- double array
 

```
if (size == maxSize){
    int *tmp = new int[2*maxSize];
    ...
}
```
- What is the cost of
 

```
vector a;
for (int i = 0; i < N; i++)
    a[i] = 0;
```

2/20/99 257

## Dynamic Array Analysis

- Count the number of arrays allocated

- Increment by one:
  - $1 + 2 + 3 + 4 + \dots + n = O(N)$  new int[ ] operations
- Double size (assume n is a power of 2)
  - $1 + 2 + 4 + 8 + 16 + \dots + n/4 + n/2 + n = O(\log N)$  new int[ ] operations

2/20/99 258

## Dynamic Array Analysis

- Count the sizes of the arrays allocated

- Increment by one:
  - $1 + 2 + 3 + 4 + \dots + n = O(N^2)$
- Double size (assume n is a power of 2)
  - $1 + 2 + 4 + 8 + 16 + \dots + n/4 + n/2 + n = 2N - 1 = O(N)$

2/20/99 259

## Printing a list in reverse order

- Iterative

```
L.GoToEndOfList();
while (! L.StartOfList()){
    L.Previous();
    cout << L.Data();
}
```

- $O(N^2)$  since Previous is  $O(N)$

- Recursive

```
void List::RevPrint(){
    if (EndOfList()) return;
    int d = Data(); Advance();
    RevPrint(); cout << d;
}
```

- $O(N)$ ,  $N$  recursive calls at  $O(1)$  each

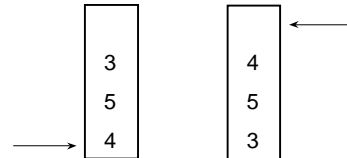
2/20/99 260

## Stack implementation

- Measure cost of inserting  $N$  elements

- Array implementation

- Insert at bottom (dumb) vs. insert at top
- Insert at bottom:  $O(N)$  to insert element since everything is copied
- Insert at top is  $O(1)$  per operation



2/20/99 261

## Fibonacci numbers

- $F(0) = F(1) = 1$

- $F(N) = F(N-1) + F(N-2)$

- 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...

- Recursive

```
int fib(int n){
    return (n <= 1) ? 1 : fib(n-1) + fib(n-2);
}
```

- Iterative

```
f[0] = f[1] = 1;
for (int i = 2; i <= n; i++){
    f[i] = f[i-1] + f[i-2];
}
```

2/20/99 262

## Sorting and Searching

- Sorting

- Insertion sort  $O(N^2)$
- Merge sort  $O(N \log N)$

- Searching - find a value in a sorted array

- Linear search  $O(N)$
- Binary search  $O(\log N)$
- 20 questions: I am thinking of a number between 1 and 1000, can you determine the number with 20 questions? (how about 15 questions, 10 questions, 5 questions)

2/20/99 263

## Summary

- Measuring Efficiency

- Measure as a function of input size  $N$
- Use steps of time or units of memory for measurements

- Asymptotic complexity

- Order of common complexity functions

- Big-O notation

- Methods for analyzing programs

2/20/99 264