

## CSE 143

### Program Efficiency

[Chapter 9, pp. 390-401]

3/25/2001 U-1

### What is Efficiency?

- Efficiency == effective use of resources
- What is a "resource"?
  - Time
  - Space or memory
  - Programmer
  - Network bandwidth
  - Others?
- We'll focus on time, but all of these can be analyzed. (Even programmers?)

3/25/2001 U-2

### Does Efficiency Matter?

- Yes! Faster is better
  - Assuming correctness, etc.
- How can we achieve faster code?

3/25/2001 U-3

### How to Speed Up Code

- Wait for the machines to get faster
- Write "tighter" code. (Gross hacks?)
- Use "better" algorithms and data structures. (These two really go together, as we'll see.)

3/25/2001 U-4

### Objective

- To convince you that the most important way to speed up a program through "better" algorithms.
- To give you some tools by which you can figure out what a better algorithm is.

3/25/2001 U-5

### Faster Machines

- Moore's law states that computers double in speed every 18 months.
  - This has held fairly true for decades
- Why not just wait for computers to get faster?
  - When might this work?
  - When might this not work?

3/25/2001 U-6

## Fast Code

- C/C++ language/culture encourages tricky coding, often in the name of "efficiency"

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

- Reasons for caution
  - Correctness?
  - Code used by others
  - No need to do compiler's job
  - 90/10 principle

3/25/2001 U-7

## Measuring Time Efficiency

- One way of measuring speed is to run the program
  - see how long it takes
  - see how much memory it uses
- Lots of variability when running the program
  - What input data?
  - What hardware platform?
  - What compiler? What compiler options?
- Just because one program runs faster than another right now, will it always be faster?

3/25/2001 U-8

## 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
- Will this still make any sense?

3/25/2001 U-9

## Complexity Analysis (2)

- Measure time and space in terms of the *size* of the input rather than details of the specific input
  - Our results will not give us absolute run times
  - We will get functions that describe how the program will slow down as the problem size grows
- Allows us to focus on big issues, and fundamental differences between algorithms
  - Don't panic—we'll see some examples!

3/25/2001 U-10

## Example For Analysis

```
// 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?

3/25/2001 U-11

## 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]

3/25/2001 U-12

## 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
}

```

- 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:  $f(N) = 5N + 3$ .

3/25/2001 U-13

## How $5N+3$ Grows

- The  $5N+3$  analysis gives an estimate of the true running time for different values of N:

N = 10 => 53 steps

N = 100 => 503 steps

N = 1,000 => 5,003 steps

N = 1,000,000 => 5,000,003 steps

- As N grows, the number of steps grows in *linear* proportion to N, for this Sum function

3/25/2001 U-14

## Methodology

- The example was typical
- 1. Analyze a program by counting steps
- 2. Derive a formula, based in some parameter N that is the size of the problem
  - For example, one algorithm might have a formula of  $N^2$
  - Another might be  $2^N$
- 3. Study the formula to understand the overall efficiency

3/25/2001 U-15

## 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	$\sim 10^{76}$
10000	13	50000	$10^5$	$10^8$	$\sim 10^{3010}$

3/25/2001 U-16

## Isn't This Totally Bogus?

- Need to run faster? Buy a faster computer!
  - Recall Moore's law
- Suppose we could make the CPU 1,000,000 times faster -- how much would that help?
  - Suppose the algorithm has complexity  $2^N$ ?
  - See following chart

3/25/2001 U-17

## If We Sped Up the CPU...

Even speeding up by a factor of a million,  $10^{3010}$  is only reduced to  $10^{3004}$

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	$\sim 10^{76}$
10000	13	50000	$10^5$	$10^8$	$\sim 10^{3010}$

3/25/2001 U-18

## How long is a Computer-Day?

If my program needs  $f(N)$  microseconds to solve some problem, how big a problem can I solve in a day?

What if I get a million times faster computer ?

$f(N)$	N for 1 day	million x, N for 1 day
N	$N = 9 \times 10^{10}$	million times larger
$5N$	$N = 2 \times 10^{10}$	million times larger
$N \log_2 N$	$N = 3 \times 10^9$	60,000 times larger
$N^2$	$N = 3 \times 10^5$	1,000 times larger
$N^3$	$N = 4 \times 10^3$	100 times larger
$2^N$	$N = 36$	+20 larger

3/25/2001 U-19

## Big numbers

- Suppose a program has run time proportional to  $n!$
- Suppose the run time for  $n = 10$  is 1 second
- Do the math:
  - For  $n = 12$ , the run time is 2+ minutes  
The time for 12 is  $12! = 10! \times 11 \times 12$  which is 132 times longer than 1 second: 132 seconds
  - For  $n = 14$ , the run time is 6 hours  
 $11 \times 12 \times 13 \times 14$  times longer
  - 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

3/25/2001 U-20

## What Matters in the Long Run?

- 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; different computers by and large differ by a constant factor
- What is fundamental is that the time is *linear* in  $N$ 
  - We say " $5N+3$  grows like  $N$ ", or " $5N+3$  is *asymptotically linear*" or " $5N+3$  is asymptotically bounded by  $N$ ", etc.

3/25/2001 U-21

## Asymptotic Complexity

- Asymptotic: what happens as  $N$  gets large
  - Focus on the highest-order term
    - Drop lower order terms such as  $+3$
  - Drop the constant coefficient of the highest order term
- This gives us an approximation of the complexity of the algorithm
  - Ignores lots of details, concentrates on the bigger picture

3/25/2001 U-22

## 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)

3/25/2001 U-23

## 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  $N$ .

3/25/2001 U-24

## 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:
  - $5N + 3 = O(N)$   
 also true:  $5N + 3 = O(N^2)$
  - $37N^5 + 7N^2 - 2N + 1 = O(N^5)$

3/25/2001 U-25

## Computer Science Footnote

- There's a whole big theory of algorithmic complexity
- Typical questions:
  - What is the worst case performance (upper bound) of a particular algorithm?
  - What is the average case performance of a particular algorithm?
  - What is the best possible performance (lower bound) for a particular type of problem?
- Many difficult questions
  - Complicated mathematics
  - Still many unsolved problems!

3/25/2001 U-26

“Computer Science is no more about computers than astronomy is about telescopes.”

-- E. W. Dijkstra

3/25/2001 U-27

## Common Orders of Growth

Let  $N$  be the input size

$O(1)$	Constant Time
$O(\log_b N) = O(\log N)$	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

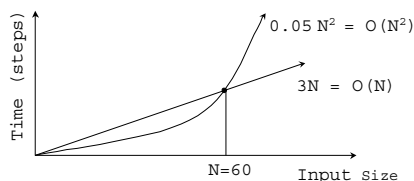
Increasing Complexity

Any integer is called “polynomial” time  
 Rule of thumb: if it ain't polynomial, it ain't practical

3/25/2001 U-28

## 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



3/25/2001 U-29

## Big-O Arithmetic: Simplified

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

3/25/2001 U-30

## 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=3;`
- Most conditional tests  
`if ( x < 12 ) ...`

3/25/2001 U-31

## 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?`  
`myList s1 = s2 + s3; // overloaded + ??`
- Declaring complex data types that have constructors
- Function invocations  
`if ( aPriorityQueue.Size() < 10 ) ...`

3/25/2001 U-32

## Analyzing Loops

Any loop analysis has two parts:

1. How many iterations are performed?
2. 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  $N \cdot O(1) = O(N \cdot 1) = O(N)$

3/25/2001 U-33

## 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  $N \gg 100$ .

3/25/2001 U-34

## 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

3/25/2001 U-35

## Nested Loops –Easy Case

- 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 += 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
  - and this does not depend on the outer loop time
- Total is  $N \cdot O(N) = O(N \cdot N) = O(N^2)$

3/25/2001 U-36

## Nested Loops – Harder Case

- 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
- For this example, 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  $O(N^2)$
- In general, finding a formula can be hard

3/25/2001 U-37

## 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  $O(N^2) + O(N) + O(1) = O(N^2)$

3/25/2001 U-38

## 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  $O(n^2)$ .

3/25/2001 U-39

## "Worst-Case" vs "Average-Case"

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

- If you knew how often the condition is true, you could compute a **weighted average**.
  - Extreme case: the conditional might be always true or never true
- “**Average case**” analysis can be very difficult
  - Use tools from probability and statistics
- For many algorithms, it is useful to know both the **worst case** and the **average case complexity**

3/25/2001 U-40

## Cost of Function Calls

$F(b, c);$

- Cost =**

```
cost of making the call
+ cost of passing the arguments
+ cost of executing the function
```

- Making and returning from the call:  $O(1)$
- Passing the arguments: depends on how they are passed
- Cost of execution: must do analysis of the function itself

3/25/2001 U-41

## Efficiency in Parameter Passing

- Pass by value -- copies entire structure
  - `Page::Translate(CodeBook cb);`
  - What if there's a copy constructor?
- Pass by reference -- does not copy, but allows updates
  - `Page::Translate(CodeBook& cb);`
  - `Page::Translate(CodeBook * cb);`
- const reference -- pass by reference, but do not allow changes
  - `Page::Translate(const CodeBook& cb);`
- Which technique should you use??

3/25/2001 U-42

## 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);
}
```

$O(1)$

- The running time is  $O(n)$ :  
n recursive calls until base case is reached, and the work done at each call is  $O(1)$
- In general, a "recurrence relation" results from the analysis, solvable with tools from math.

3/25/2001 U-43

## 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?
- Recurrence relation:  $T(n) = T(n-1) + T(n-2) + O(1)$
- Running time? Solve the equation

3/25/2001 U-44

## List Implementations

N is the list size

	array	linked list	doubly linked list
constructor			
isEmpty			
isFull			
reset			
advance			
endOfList			
data			
size			
insertBefore			
insertAfter			
deleteItem			

3/25/2001 U-45

## 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	$O(N)$	$O(N)$	$O(1)$

3/25/2001 U-46

## Dynamic arrays

- When array is full, reallocate new array

- strategy 1: increase size by one

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

- strategy 2: double array size

```
if (size == maxSize){
    int *tmp = new int[2*maxSize];
    ...
}
```

- What is the cost of

```
Vector a;
for (int i = 0; i < N; i++)
    a.insert(i, i);
```

3/25/2001 U-47

## 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)

- $1 + 2 + 4 + 8 + 16 + \dots + n/4 + n/2 + n = 2N - 1 = O(N)$

3/25/2001 U-48



## 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

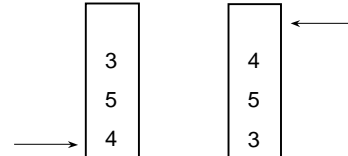
3/25/2001 U-49

## 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



3/25/2001 U-50

## Review: Common Orders of Growth

Memorize!

$O(k) = O(1)$	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	

Nanyinteger is called "polynomial" time

Rule of thumb: if it ain't polynomial, it ain't practical

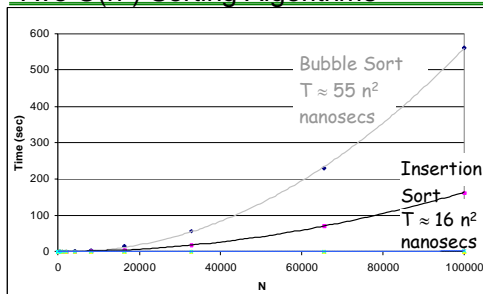
3/25/2001 U-51

## Summary So Far

- Measuring Efficiency
  - Measure as a function of input size  $N$
  - Use steps of time or units of memory for measurements
- Asymptotic complexity
  - Growth rate as  $N$  gets large
- Order of common complexity functions
- Big-O notation
- Methods for analyzing programs
- Complexity of list, stack, etc. implementations

3/25/2001 U-52

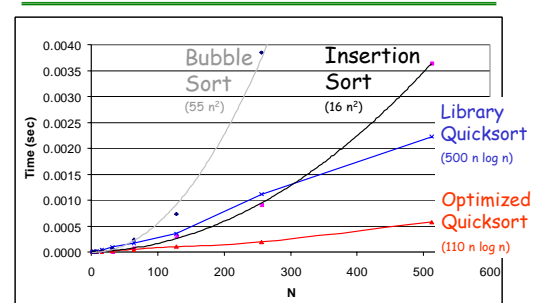
## Do Constants Matter? Two $O(n^2)$ Sorting Algorithms



Benchmark run on 233Mhz P II, 96M, NT 4.0, VC 6.0, 5/30/00

3/25/2001 U-53

## $O(n \log n)$ vs $O(n^2)$ : closeup



3/25/2001 U-54

## Summary

---

- FIRST pick the right algorithm
  - Big-O helps do that
  - Can give *many* orders of magnitude improvement
- THEN optimize it
  - above 2x improvement is uncommon

Premature optimization is the  
root of all evil -- D. Knuth

3/25/2001 U-55