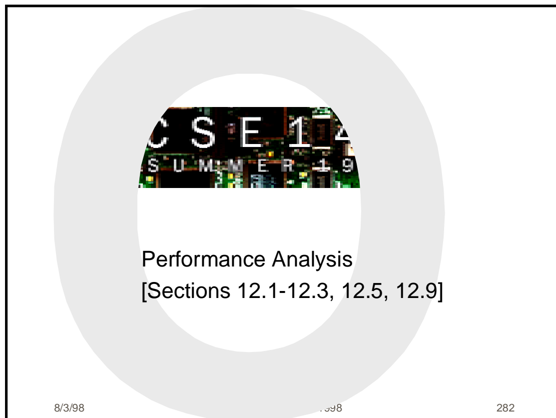


Performance Analysis

[Sections 12.1-12.3, 12.5, 12.9]

Why We Need Performance Analysis

- Every problem has many possible solutions
  - Example: Sorting

selection sort, insertion sort, bubble sort, two-way bubble sort, tree sort, heap sort, quicksort, shell sort, merge sort, radix sort, bucket sort…

- How do we know which one is best?
- We'd like some generic basis for comparison

What to Measure

- Performance can mean different things:
  - **Space efficiency**: how much memory does the program use?
  - **Time efficiency**: how long does the program take to run?
  - **User efficiency**: how easy is it to use the program?
- We'll focus on time efficiency
- Remember: correctness is more important than performance!

Benchmarking

- One solution is to use benchmarking
  - Pick a standardized problem or set of problems
  - Run every algorithm on the same problem set using the same machine
- Results are very sensitive to choice of problems, hardware, compiler, compiler options, load, …
- Results don't show how performance changes with size of problem
- Good tool for comparing different machines

Program Analysis

- When comparing algorithms, we use mathematical program analysis techniques
  - Mathematical analysis of source code
  - Independent of any machine/memory/compiler details
  - Gives very general idea of algorithm's time complexity
  - Shows how running time depends on problem size
  - Asymptotic analysis: don't care about a small number of irregularities
- An entire branch of Computer Science!

Kinds of Analysis

- Best-case analysis: how fast does the program perform in the best case?
- Worst-case analysis: how fast does it perform in the worst possible case?
- Average-case analysis: what's the average performance across all possible inputs of a given size?
- We'll stick to worst-case analysis

Big-O Notation

- Definition: for functions $f(n)$ and $g(n)$, we write

  $$f(n) = O(g(n))$$

  And say "f is big oh of g" when a constant multiple of g is eventually always bigger than f.
- Mathematically: $f(n)=O(g(n))$ means that there exists an integer $N>0$ and real number $c>0$ such that for all $n>N$, $f(n)<=cg(n)$
- Goal: make g as simple and small as possible

Huh?

- When analyzing performance, we create a function that represents the running time of the algorithm
- It's hard to define the function precisely
- So we *approximate* the running time by finding a reasonable big-O bound for the function
  - "The running time of BlargSort is $O(n^3)$"
- Comparing big-O bounds gives us a good sense of performance

Common big-O Bounds

- Let k be any fixed constant

$O(k) = O(1)$	Constant Time
$O(\log_k n) = O(\log n)$	Logarithmic Time
$O(n)$	Linear Time
$O(n \log n)$	n log n time
$O(n^2)$	Quadratic Time
$O(n^3)$	Cubic Time
$O(n^k)$	Polynomial Time
$O(k^n)$	Exponential Time

Big-O Arithmetic

- Figure 12.2, page 546
- Rules to remember:

$O(k*f) = O(f)$ (constants get absorbed)

$O(f+g) = \max(O(f), O(g))$ ("lower-order terms" get absorbed)

$O(f*g) = O(f)*O(g)$

Analyzing Simple Statements

```
int c = a / b;
```

```
double r = s * t + 16.9 / s;
```

```
int c = a / b;
int d = a * b;
int e = c + d;
```

```
Fraction f1( 1 );
Fraction f2( 1, 43 );
Fraction f3 = f1 + f2;
```

```
IntStack s2 = s1;
```


Analyzing Loops

- Two questions
  - How many times does the loop execute?
  - What's the performance bound of each iteration?
- Sometimes have to add in performance of set-up and condition, too

```
int sumArray( int data[], int n )
{
    int total = 0;
    for( int idx = 0; idx < n; ++idx ) {
        total += data[ idx ];
    }
    return total;
}
```

Bounded Loops

- What's the big-O bound of this function?

```
int sumArray100( int data[] )
{
    int total = 0;
    for( int idx = 0; idx < 100; ++idx ) {
        total += data[ idx ];
    }
    return total;
}
```


Nested Loops

- Recursively apply the rule for loops!

```
int countSevens( int data[][], int n )
{
    int total = 0;
    for( int idx = 0; idx < n; ++idx ) {
        for( int jdx = 0; jdx < n; ++jdx ) {
            if( data[ idx ][ jdx ] == 7 ) {
                ++total;
            }
        }
    }
    return total;
}
```


Tricky Nested Loops

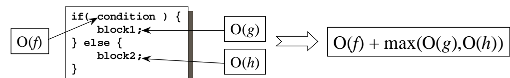
- Sometimes, the inner loop bound depends on the outer index:

```
int triangle( int n ) {
    int sum = 0;
    for( int idx = 0; idx < n; ++idx ) {
        for( int jdx = 0; jdx < idx; ++jdx ) {
            sum += idx * jdx;
        }
    }
    return sum;
}
```

- Must analyze both loops together
- What's the total number of iterations?

Conditionals

- Only one branch of a conditional will ever be evaluated
- Since we're doing worst-case analysis, we're pessimistic



Analyzing Recursion

- The most difficult kind of performance analysis
- Two questions:
  - How many recursive calls are there?
  - How much work is done at each step in the recursion?
- Example

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


The Other Canonical Computer Science Function

- Fibonacci numbers:

$F_1 = 1$
$F_2 = 1$
$F_n = F_{n-1} + F_{n-2}$

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 143, …

- Problem 1: write a recursive function to compute the nth Fibonacci number
- Problem 2: What's the big-O bound of your function?
- Problem 3: Can you do better?

Sample Use of Performance Analysis

- Compare `IntStack::push()` in the array implementation vs. the linked-list implementation
- Array implementation shifts all elements over to make room for new item: $O(n)$
- Linked-list implementation simply adds on a new box: $O(1)$
- Which would you choose?

Summary

- Different notions of efficiency
- Different kinds of time efficiency
- Best-case, worst-case, average-case
- Big-O notation
  - Arithmetic with Big-O
  - Commonly used bounds
  - Efficiency as a function of size of input
- Techniques for finding bounds on common program structures