CSC 143 Java

Program Efficiency &
Introduction to Complexity Theory

Program Efficiency & Resources

- Question: Given different implementations, which one is "better?"
- Goal: Find way to measure resource usage in a way that is independent of particular machines/implementations
- Resources
  - Execution time
  - Execution space (choosing the correct data structure)
  - Network bandwidth
  - others
- We will focus on execution time
  - Basic techniques/vocabulary apply to other resource measures

Analysis of Execution Time

- First: describe the ***size*** of the problem in terms of one or more parameters
  - Often size of data structure, but can be magnitude of some numeric parameter, etc.
- Then, count the number of **steps** needed as a **function of the problem size**

Cost of operations: Constant Time Ops

- Constant-time operations: each take one abstract time "step"
  - Simple variable declaration/initialization (double sum = 0.0;)
  - Assignment of numeric or reference values (var = value;)
  - Arithmetic operation (+, -, *, /, %)
  - Array subscripting (a[index])
  - Simple conditional tests (x < y, p != null)
  - Operator new itself (not including constructor cost)

    Note: new takes significantly longer than simple arithmetic or assignment, but its cost is independent of the problem we're trying to analyze
- Note: watch out for things like method calls or constructor invocations that look simple, but are expensive

Costs of Statements

- Sequential: S1; S2; … Sn

  sum the costs of S1 + S2 + … + Sn
- Conditional: how long it *might* take to execute the code

```
if (condition) {  // take max cost ( S1, S2) (plus cost of evaluating the condition)
    S1;
} else {
    S2;
}
```
- Loop:

  Calculate cost of each iteration

  Calculate number of iterations

  Total cost is the product of these

Example

- What is the running time of the following logic?

```
//Given some array vector of size N
double ans = 0.0;
for (int k = 0; k < N; k++) {
   ans = ans + vector[k];
}
```
- What things happen only once?
- What things happen N times?
- Add them all up

Function Calls

- Cost for f(a, b, c) is
  - Cost of actually **calling** the function (constant overhead)
  - + cost of **evaluating** the arguments
  - + cost of **parameter passing** (normally constant time in Java for both numeric and reference values)
  - + cost of **executing** the function body

Exercise

- Analyze the running time of printMultTable
  - Pick the problem size
  - Count the number of steps

```
// print multiplication table with
// n rows and columns
void printMultTable(int n) {
    for (int k=1; k <=n; k++) {
        for (int c = 1; c <=n; c++ ) {
            System.out.print(r * c + " ");
        }
        System.out.println();
    }
}
```


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 + … + (N-1)==(N)(N-1)/2`

Comparing Algorithms

- Suppose we analyze two algorithms and get these times
  - Algorithm 1: $2n^2 + 37n + 120$
  - Algorithm 2: $5n + 3$

How do we compare these? What really matters?

- Answer: In the long run, the thing that is most interesting is ***the cost as the problem size n gets large***
  - What are the costs for n=10, n=100; n=1,000; n=1,000,000?
  - Mainstream computers are so fast these days that time needed to solve small problems is rarely of interest

Not necessarily so for slow, low-power, or embedded systems

Orders of Growth. Does it matter?

What happens as the problem size doubles?

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	$\sim10^9$
64	6	320	384	4096	$\sim10^{19}$
128	7	640	896	16384	$\sim10^{38}$
256	8	1280	2048	65536	$\sim10^{76}$
10000	13	50000	10^5	10^8	$\sim10^{3010}$

Asymptotic Complexity

- Asymptotic Complexity: As N gets large, focus on the highest order term
  - Ignores lots of details, concentrates on the bigger picture
- The asymptotic complexity gives us a (partial) way to answer "which algorithm is more efficient"
  - Algorithm 1: $2n^2 + 37n + 120$ is proportional to n^2
  - Algorithm 2: $50n + 42$ is proportional to n
- Graphs of functions are handy tool for comparing asymptotic behavior

Big-O Notation

- Definition: If f(n) and g(n) are two complexity functions, we say that

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

  if there is a constant c such that

  $f(n) \le c \cdot g(n)$

  for all sufficiently large n

pronounced f(n) is Big-O(g(n)) or is order g(n)

Implications

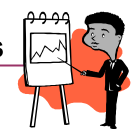
- The notation f(n) = O(g(n)) is **not** an equality; think of it as shorthand for
  - "f(n) grows at most like g(n)" or
  - "f grows no faster than g" or
  - "f is bounded by g"
- O( ) notation is a *worst-case* analysis
  - Generally useful in practice
  - Sometimes want *average-case* or *expected-time* analysis if worst-case behavior is not typical (but often harder to analyze)

Complexity Classes

- Several common complexity classes (problem size n) Memorize these!
  - Constant time: O(k) or O(1)
  - Logarithmic time: O(log n) [Base doesn't matter. Why?]
  - Linear time: O(n)
  - "n log n" time: O(n log n)
  - Quadratic time: $O(n^2)$
  - Cubic time: $O(n^3)$

    …
  - Exponential time: $O(k^n)$
- $O(n^k)$ is often called *polynomial time*
- Rule of thumb: polynomial time = practical; exponential time = find a different algorithm

Analyzing List Operations



We can use O( ) notation to compare the costs of different list implementations

Operation	Dynamic Array	Linked List*
Construct empty list		
Size of the list		
isEmpty		
Clear		
Add item to start of list		
Locate item (contains, IndexOf)		
Add or remove item once it has been located		

*** with head, tail, and size**

Practical Advice For Speed Lovers

- First pick the right algorithm and data structure
  - Implement it carefully, insuring correctness
- Then optimize for speed – but only where it matters
  - Constants do matter in the real world
  - Clever coding can speed things up, but the result is likely to be harder to read, modify
  - Use tools to find hotspots – concentrate on these

> *"Premature optimization is the root of all evil"*
>
> – Donald Knuth

More Advice…

> *"It is easier to make a correct program efficient than to make an efficient program correct"*
>
> **-- Edsgar Dijkstra**

Summary

- Analyze algorithm sufficiently to determine complexity
- Compare algorithms by comparing asymptotic complexity
- For large problems, an asymptotically faster algorithm will always trump clever coding tricks
- Optimize/tune only things that actually matter, once you've picked the best algorithm

Computer Science Note

- Algorithmic complexity theory is one of the key intellectual contributions of Computer Science
- Typical problems
  - What is the worst/average/best-case performance of an algorithm?
  - What is the best complexity bound for all algorithms that solve a particular problem?
- Interesting and (in many cases) complex, sophisticated math
- Still some key open problems