Lecture 4
Jan 29 2007
Shlomo Hershkop
Announcements

- Hope you had a chance to attend a review session
- Linked off main class page
  - Moved to resource page (makes more sense)

Reading:
- Skim chapter 1
- Read chapter 2
Reviewing

- Make sure you are comfortable writing Java code
  - If you missed the review go over slides and attempt to do the programming

- Please speak to me if you need more help

- See Chapter 1 for more background on java
Abstraction

- Generally when we program anything complex (yes more complicated than hello world) need to think about what you want to do

- Separate how to use the class

- and how we are representing the information which the class is representing
definition

- Data structure
  - Systematic way of organizing and accessing some data

- Algorithm
  - Step by step procedure for performing some task in a finite amount of time
So how do you measure how “GOOD” a DS or algorithm is?
Generally Speaking

- Analysis Stage – we will try to calculate goodness

- We will be looking at how fast something might run

- Will look at much space we will need to run the algorithm
  - Can have other measures also

- Want to see how things change (better/worse) as input increases
Testing by experiments

- In theory we could simply run the algorithm over a set of inputs to see how good it is
  - This won’t work because...
    - Inputs might not be “real” enough
    - Hard to replicate to other algorithm unless exact set of hardware/software conditions
    - Programming language will affect the runtime that we see
Model of Computation

- In order to analyze algorithms
  - Will want to consider a model to study what it means to compute across all possible inputs
  - would like to create classes of algorithms, so that we can talk about them in a uniform way independent of hardware/software
    - broad categories
    - High level
  - Will make some simplifications in our theoretical model
Things to consider

- **Computation**
  - Assume every step of the algorithm takes one step
  - Different than real life
    - Generally Addition/subtraction < Multi <<< Division
    - CPU tasks << Memory access <<<<<< Disk access
  - Will come back to this when we discuss multi threaded environments

- **Space**
  - Assume infinite memory
  - Will adjust later

- **Time**
  - Will be counting time steps
Simplifying analysis

- Lets focus the analysis of how an algorithm is doing by focusing on how the run time grows as input increases
  - Lets use a function to map set of inputs to running times
Basic Model – Theta bound

- $T(N) = \Theta(g(N))$
  - set of functions $f(N)$ are in $\Theta(g(N))$

  if there exists positive constants $c_1, c_2, n_0$

  such that $0 < c_1 \cdot g(N) < f(N) < c_2 \cdot g(N)$

  for all $N \geq n_0$
- Theta bound is strongest bounding
- Real world sometimes hard to make such guarantees
- Need to relax bound
Big-O

- $T(n) = O(g(N))$

if there are positive constants $c$ and $n_o$ such that $T(N) \leq c g(N)$ when $N \geq n_o$

- Known as Big O notation

- So the relationship is asymptotic in the upper limit
Omega

- lower bound only
little $o$

- little $-o$ provides an upper bound but not a tight one
- doesn’t say much
- should be aware of it, but won’t be using it much
Summary

- We would like to use functions to describe the growth of some resource by an algorithm.
- Want to compare different algorithms by growth rate.
- Big O allows us to define an upper bound on a function.
- So we can say: something is on the order of Big-O of something else.
Careful

- On small input sizes, it is hard to analyze an algorithm
- Might be lucky

- Its been shown time and time again that something which just “works” but poorly designed can have some very expensive ramifications when scaling goes up
Example

- Suppose we have two algorithms to solve the same problem
  - Algorithm A
    - $O(n)$
  - Algorithm B
    - $O(n^2)$

- Which is asymptotically better?
Simplification

- Say an algorithm is said to run as the following function of the input:
  \[ f(n) = 3n^2 + 2n + 5 \]

- When considering the Big O runtime
  - Drop constants
  - Drop low order polynomial terms
  - We are interested in the function as it is taken to the limit

- This simplification can come back to haunt you if you completely ignore the stuff you are throwing out
Analyze

- Input size is strong consideration

- Generally an algorithm might have
  - Best case (ha!)
  - Worst case
  - Average case

- Which is most interesting?
Other considerations

- Remember it’s a great tool, but very simplified

- Take into consideration:
  - Programming language
  - Compiler
  - Computer code
Example to analyze

```java
public int sum(int n) {

    int part_sum = 0;

    for(int i=0;i <= n; i++){
        part_sum += i * i * i;
    }

    return part_sum;
}
```

- For simplification next let us review some general rules
Rules

- Consecutive statements
  - Just add consecutive statements within a code block

- Loops
  - Running time of a loop is at most the running time of statements inside (plus tests) multiplied by number of iterations
Nested Loops

- Analyze inside out

```php
for ( $i = 0; $i < $n; $i++ )
{
    for( $j = 0; $j < $n; $j++ )
    {
        k++;
    }
}
```
Example

```php
for( $i = foo_1(); $i < $n; $i++)
{
    somesub($i);
    $total += foo2();
}
```
More Rules

- **If/else**
  - The runtime of if/else is the test plus the larger of the running time
  - Take worst behavior
Example 1

```c
int findMax(int list[], int max) {
    int maxValue = list[0];
    for (int i = 0; i < max; i++) {
        if (maxValue < list[i]) {
            maxValue = list[i];
        }
    }
    return maxValue;
}
```
int Example2 (int list[], int max) {
    int k = 0;

    for (int i = 0; i < max; i++) {
        for (int j = 0; j < max; j++) {
            k = (i * j) + n;
        }
    }

    return k;
}
Example 3

```c
int Example3(int n) {
    int k = 0;
    for (int i = 0; i < 1000; i++) {
        k = k + n;
    }
    return k;
}
```
Example 4

sub Example4(int n) {
int k =0;

    for( int i=0; i < n; i++) {
        for(int j =0; j < n*n; j++) {
            k = (i * j) + n;
        }
    }

return k;
}
Example 5

```c
int Example5(int n) {
    int k = 0;
    for(int i = 0; i < n; i++) {
        for(int j = 0; j < n; j++) {
            k += Example4(n);
        }
    }
    return k;
}
```
Example 6

```c
int Example6(int n) {
    int k = 0;
    while (n > 1) {
        n -= 1;
        k++;
    }
    return k;
}
```
Example 7

```c
int Example7( int n) {
  int k =0;
  while (n > 1) {
    n = n / 2;
    k++;
  }
  return k;
}
```
Example 8

```c
int Example8 (int n) {
    if(n == 0 ) {
        return 1;
    } else {
        return Example8(n/2) + 1;
    }
}
```
Next time

- Read chapter 2, start 3.1