eecs 281

Data Structures and Algorithms

Discussion 2

MichiganEngineering

Contents

- Gprof
- Revision Control
- Complexity Analysis
- Recurrence Relations
- Debugging: Final Comments
- Project 1

Gprof

Gprof: Motivation

- Program is running slow.. where is the bottleneck?
- What functions are called more or less than others?

What have you used in the past to find bottlenecks?

Gprof: Introduction

- Runtime profiling tool

Goal is to improve code (and coding) efficiency by devoting time optimizing only the *most important* parts of code

Gprof: Introduction

- How do we decide which parts of our code are the most worthwhile to optimize?
  - Function that runs the most times?
  - Function that runs the slowest?
  - Function that consumes the most total time?

Gprof: General Usage

1. Specify the **`-pg`** option when compiling code
2. Run the executable as normal to generate profile data
   a. Creates a file called 'gmon.out' in local directory
   b. Keeps track of function calls and monitors runtime performance
3. Run: `gprof [options] [exe name] > output.txt`

Gprof: Example

Gprof: Output

- Creates 2 structures:
  - Flat profile (-p option)
  - Call graph (-q option)

Gprof: Output (Flat Profile)

- Flat Profile

% time	cumulative seconds	self seconds	calls	self us/call	total us/call	name
100.00	0.10	0.10	10000	10.11	10.11	slow_code(int)
0.00	0.10	0.00	10000	0.00	0.00	fast_code(int)

Gprof: Output (Call Graph)

- Call Graph

```
index % time    self  children    called     name
                                                 <spontaneous>
[1]    100.0    0.00    0.10                   main [1]
                0.10    0.00   10000/10000        slow_code(int) [2]
                0.00    0.00   10000/10000        fast_code(int) [7]
-----------------------------------------------
                0.10    0.00   10000/10000        main [1]
[2]    100.0    0.10    0.00   10000           slow_code(int) [2]
-----------------------------------------------
                0.00    0.00   10000/10000        main [1]
[7]      0.0    0.00    0.00   10000           fast_code(int) [7]
-----------------------------------------------
```

Gprof: Caveats

- Statistical accuracy
  - Do <u>not</u> interpret run times as absolute truth.
  - Run time will vary slightly from run to run
- Times are summed across all calls to that function.
  - Could be that the function is highly optimized for all but one set of inputs.
- Mutual Recursion A-->B-->A-->B
  - Makes the call graph more difficult to read by separating into cycles

Revision Control

Revision Control: Motivation

- You submit code to the autograder and receive an 80%
- You make a few changes to your code, and now when you submit, you receive a 60%
- You want to revert back to your old code, but you forgot the changes you made…
- Crap.

Revision Control: Motivation

- You work on a program that has over 1000 files associated with it on a North Campus computer
- Since you plan on working on the program later on Central Campus, you decide to zip your files and send them to yourself via email
- …Except your files are too big to send via one email, so you have to send 10…
- Crap.

Revision Control: Types

- Git
  - Central repo with many client repos (each with user)
  - Allows offline development
  - Commonly utilized via github
  - https://education.github.com/pack
  - https://www.atlassian.com/git/tutorials/
- SVN (Apache Subversion)
  - One repo with lots of clients

Important

Do not make a public repository!! If other students find your code, it will be considered an honor code violation!

How to get a free private repo:

- https://education.github.com/pack
- https://www.atlassian.com/git/tutorials/

Complexity Analysis

Big-*Oh*: O(n)

Big-*Omega*: Ω(n)

Big-*Theta*: Θ(n)

Big-*Oh* Definitions

$$f(n) = O\big(g(n)\big) \; iff \; \exists \; c > 0, n_0 \geq 0 \; s.t.$$
$$f(n) \leq c * g(n) \; \forall \; n \geq n_0$$

or

- `c*g(n)` "upper bounds" `f(n)` at `n >=` n_0
- Example: `f(n) = n`, `g(n) =` n^2
  - Is `f(n) = O(g(n))`? What are `c` and n_0?

Big-*Omega* Definitions

$$f(n) = \Omega\big(g(n)\big)\ iff\ \exists\ c > 0, n_0 \geq 0\ s.t.$$
$$f(n) \geq c * g(n)\ \forall\ n \geq n_0$$

or

- `c*g(n)` "lower bounds" `f(n)` after `n >=` n_0
- Example: `f(n) = 4*n`, `g(n) = 0.5*n`^{2}
  - Is `f(n) =Ω(g(n))`? What are `c` and n_0?

Big-*Oh* and Big-*Omega* Bounds

- Only meaningful to state tightest bounds
  - If `n + 5 = O(n)` ← tightest upper bound
    then `n = O(`n^3`)`, `n = O(`n^{100}`)`, `n = O(`2^n`)`, etc.
    → true upper bounds, but not useful
  - If `n + 5 = Ω(n)` ← tightest lower bound
    then `n = Ω(lg n)`, `n = Ω(1)`, etc.
    → true lower bounds, but not useful

Big-*Theta* Definitions

$$f(n) = \Theta\big(g(n)\big)\ iff$$
$$f(n) = O\big(g(n)\big)\ and\ f(n) = \Omega\big(g(n)\big)$$

or

- **`g(n)`** "tightly bounds" **`f(n)`**
- Example: **`f(n) = 2*n`**^{2}, **`g(n) = 0.5*n`**^{2}
  - Is **`f(n) =Θ(g(n))`**?

Complexity Analysis

- What for?
  - Determine how well an algorithm scales for larger inputs
  - Compare performance of two algorithms
- How?
  - Create a function f(n) to express rate of growth
  - Big-Oh (O) notation
  - Let’s do an example

Complexity Analysis Example

```cpp
// what is the complexity?
int main() {
    int size = 10;
    int count = 0;
    vector<vector<vector<int> > > cube;
    for(int i = 0; i < size; ++i) {
        cout << "\n next level \n";
        vector<vector<int> > section;
        for(int j = 0; j < size; ++j) {
            vector<int> row;
            for(int k = 0; k < size; ++k) {
                row.push_back(count);
                cout << count++ << ' ';
            }
            section.push_back(row);
            cout << '\n';
        }
        cube.push_back(section);
    }

    cout << "\n\n ------ FINISHED ------ \n\n";
}
```

Complexity Analysis Example

```
void array2Dsearch(vector<vector<int> > & array2D,
         int item) {

    for(int i = 0; i < array2D.size(); ++i) {// n steps
    if(binary_search(array2D[i].begin(),
                  array2D[i].end(), item)) { // log(n)

       cout << "found " << item << '\n';  // 1 step
       return;                            // 1 step
    }
    }
    cout << "did not find " << item << '\n'; // 1 step
}
// Total: n*(log(m) + 1) + 1 = O(nlog(m)
```

MichiganEngineering

Recurrence Relations

Recurrence Relations

- Recurrence Relation: an equation that recursively defines a sequence
  - Usually used to analyze algorithm runtimes
- Many algorithms loop on a problem set, do something, and create smaller sub-problems
- To solve them: how is the problem set changing each time?

Steps to Solve RR

- Find T(n-1), T(n-2), …, T(2), T(1), T(0)
  - Sometimes, n = 2^k
- Apply base case (initial terms)
- Plug in previous term into current equation
  - EX: plug in T(n-2) equation into T(n-1) equation
- Find summation formula
- Solve summation formula (T(n))
- Find big-oh of T(n)

Iteration Method

- Try substituting backwards until a pattern is found
- Ex: T(n) = 3T(n/4) + n, T(1) = 1

- T(n) = n + 3(n/4 + 3T(n/16))
  = n + 3n/4 + 9(n/16 + 3T(n/64))
  = n + 3n/4 + 9n/16 + 27T(n/64)
  = … = n + 3n/4 + 9n/16 + … + 3^(K)*T(1)
- Obvious (¾)^n term. K = $\log_4 n$ terms to T(1).
- $$T(n) \leq n \sum_{i=0}^{\log_4 n} \left(\frac{3}{4}\right)^i + 3^{\log_4 n}$$

Master Theorem Basis

Let $T(n)$ be a monotonically increasing function that satisfies:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

$$T(1) = 1$$

Where $a \geq 1$, $b \geq 2$. If $f(n) \in \Theta(n^c)$, then:

$$T(n) = \begin{cases} \Theta(n^{\log_b a}) & \text{if } a > b^c \\ \Theta(n^c \log_2 n) & \text{if } a = b^c \\ \Theta(n^c) & \text{if } a < b^c \end{cases}$$

Exercises

- T(n) = 2*T(n-1) + 1, T(1) = 1
- T(n) = 2*T(n/2) + n, T(1) = 0

*Important Formula:

$$\sum_{i=m}^{n} a * r^{i} = \frac{a\left(r^{m} - r^{n+1}\right)}{1-r} \qquad \sum_{i=m}^{n} i = \frac{(n+m)(n-m+1)}{2}$$

Debugging: Final Comments

Final Comments

- Don't debug with print statements!
  - Is perhaps easier at first, but being comfortable with debugging tools is important in the long run
- Part of the class is learning how to debug.
  - In OH, we will expect you to have made a reasonable effort to debug the problem yourself before coming to us.
- Turn off all the debugging flags for your final submission
  - Debugging tools are great, but they can <u>severely</u> degrade your code's performance. Why?
  - Making separate Makefile commands can prevent you from forgetting

Project 1: Advice

- Write modular code
- Provide useful data structures
- Think hard about data structures
- Utilize good comments and a standard coding style
  - Readable code => better help in OH’s
- See Piazza posts often
- Submit ASAP
  - Students often take 10+ submits to get a desired score

MichiganEngineering

Project 1: Questions?

Recursion Tree Method

- Draw a picture of the back substitution process
- Ex: T(n) = 2T(n/2) + n^2, T(1) = 1