CSE 15L:
Software Tools and Techniques Laboratory

Winter 2021 - http://ieng6.ucsd.edu/~cs15x

Instructors: Gary Gillespie Keith Muller

Class sessions will be recorded and made available to students asynchronously.

Topic 1 : Measuring Software Execution Time

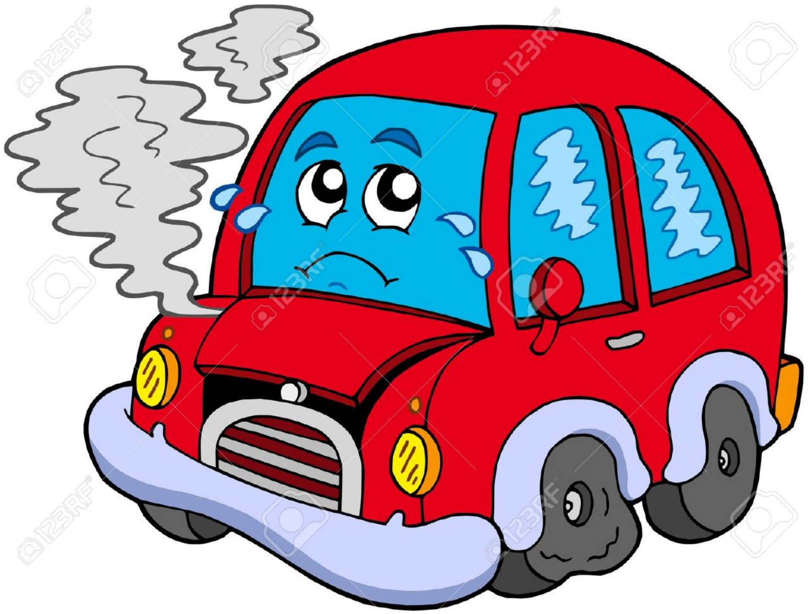
Topic 2: Diagnostic Logging

Debugging is finally done! Ready for a test drive?

My program runs, but seems very slow …

CPU
Throughput
Instruction stall

Memory
Latency
Cache misses

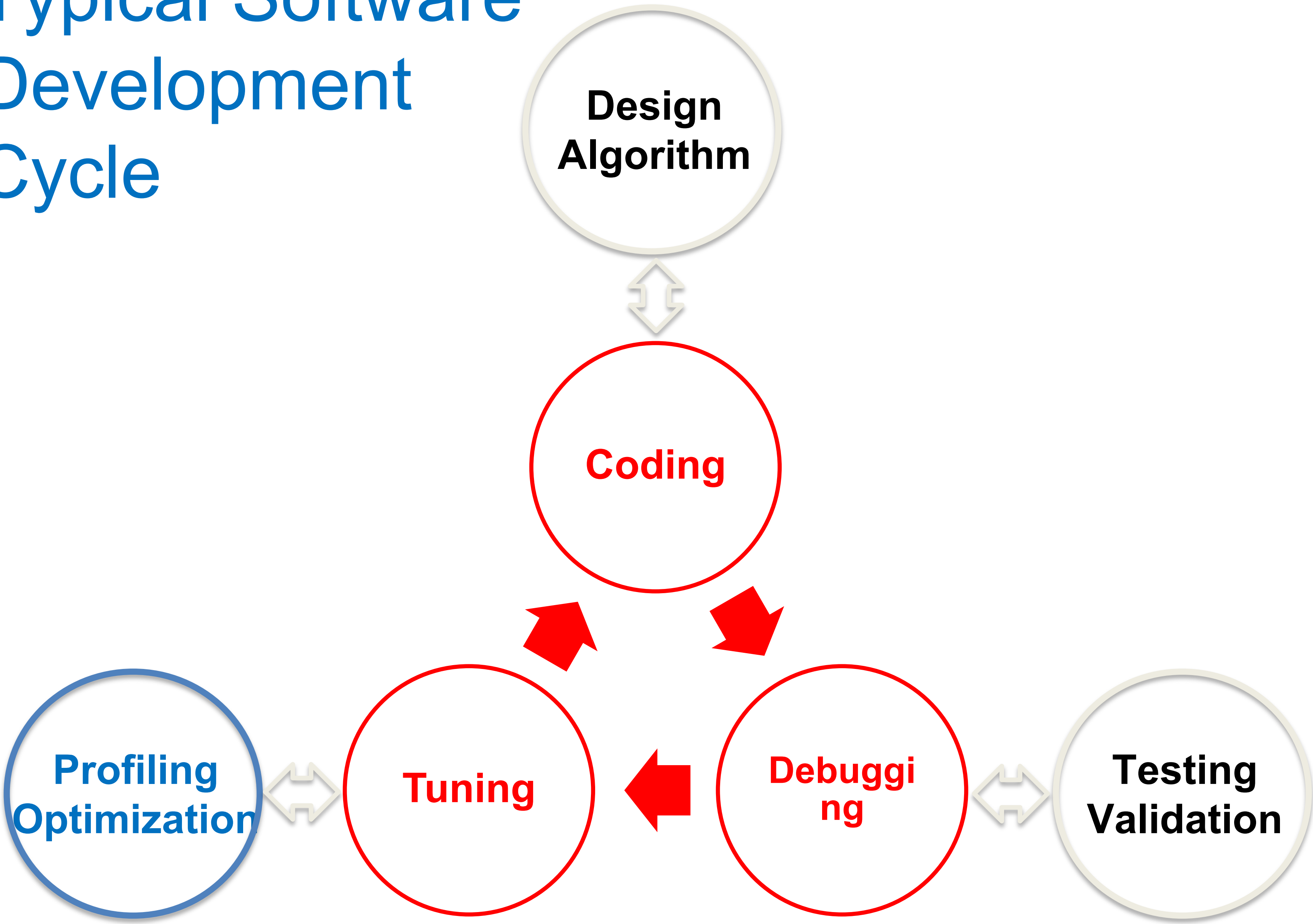


I/O
Communication
DB contention

Multithreading
Load balancing
Scalability

Where to start?

Typical Software Development Cycle



Software Cost/Performance

- Software cost-performance can be stated in terms of cost/performance

- Costs should be stated as functions of the size N of the problem the software is given to solve:
  - **T(N):** time cost function
  - **S(N):** space (memory) cost function
  - **E(N):** energy cost function *(reality is more complex as you must minimize total cost to support the software execution service)*

Commercial Optimization: From Prototype to Production

- Goal 1: Proper functionality and error free execution
- Goal 2: Scalability with increased use (do the software scale with use)
- Goal 3: Best cost/performance (delivers performance at the lowest cost –hardware related costs
- Goal 3A: Minimize stranded HW resources (resources utilized at 100% capability over 100% of time)
- Costs: Equipment (compute and infrastructure), power, floorspace, management staff

Determining Cost Functions

- You can **analyze** the software to determine cost functions: look at the source code
  - ***Time cost:*** *how many instructions will be executed*
  - ***Space cost:*** *how many variables will be created*
  - ***Energy cost:*** *usually determined by CPU and memory usage*
- Or you can **measure** the cost functions: implement the software, instrument it, and run it

Identifying Program Hot Spots

Gather statistics about your program's execution

- ***Coarse-grained:*** how much time did execution of a particular function call take?
- Time individual function calls or blocks of code
- ***Fine-grained:*** how many times was a particular function called? How much time was taken by all calls to that function?
- Use an **execution profiler** such as **gprof**

Measuring Cost Functions

- **Profiling frameworks** exist for instrumenting running software and collecting time, memory, and energy usage data in detail

- First, we will look at a simpler approach for measuring time cost: accessing the system clock, and measuring "**wall clock time**" taken to perform an operation

The Unix time commands

- Java profiling can give very fine-grained information about time costs of operations
- Another useful tool are the Unix **`time`** commands
- These will summarize time (and optionally other) costs of the entire execution of a program
- Under Linux, there are two versions: the bash shell built-in **`time`** , and **`/usr/bin/time`**

The Unix time commands

- To time a Java application using the built-in time:

  `time java MyApp`

- To time a Java application using /usr/bin/time, with verbose output:

  `/usr/bin/time –v java MyApp`

- Many other options are available for /usr/bin/time; see man time for more info

Program Under Study: prof.c

```c
#include <stdio.h>
#include <math.h>
#define NUM 100000

void doit()
{
    double x = 0;
    for (int i = 0; i < NUM; i++)
    x += sin(i);
}

void f()
{
    for (int i = 0; i < 10000; i++)
    doit();
}

void g()
{
    for (int i = 0; i < 50000; i++)
    doit();
}
```

```c
int main()
{
    double s = 0;
    for (int i=0;i<10000*NUM; i++)
        s += sqrt(i);
    f();
    g();
    return 0;
}
```

Profiling a C or C++ Program

- To time a C application using the built-in time:

`time ./prof`

```
$ time ./prof
real 6m30.358s
user 6m30.305s
sys 0m0.002s
```

Output:

- **Real**: Wall-clock time between program invocation and termination
- **User**: CPU time spent executing the program
- **System**: CPU time spent within the OS on the program's behalf

Using gprof

Step 1: Instrument the program

gcc –pg prof.c –o prof –lm

- Adds profiling code to prof, it “Instruments” **prof**

Step 2: Run the program

./prof

- Creates file ./**gmon.out** containing statistics

Step 3: Create a report

gprof ./prof > myreport

- Use **./prof** and ./ **gmon.out** to create textual report

Step 4: Examine the report

cat myreport

Using gprof – flat profile

Percentage of the total execution time your program spent in this function. These should all add up to 100%.

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  s/call   s/call  name
 69.34     38.81    38.81    60000    0.00     0.00  doit
 31.06     56.20    17.39                            main
  0.02     56.21     0.01        1    0.01     6.48  f
  0.00     56.21     0.00        1    0.00    32.34  g
```

Each line describes one function

- **name**: name of the function
- **%time**: percentage of time spent executing this function
- **cumulative seconds**: [skipping, as this isn't all that useful]
- **self seconds**: time spent executing this function
- **calls**: number of times function was called (excluding recursive)
- **self s/call**: average time per execution (excluding descendents)
- **total s/call**: average time per execution (including descendents)

This is cumulative total number of seconds the spent in this functions, plus the time spent in all the functions above this one

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 69.34     38.81    38.81    60000     0.00     0.00  doit
 31.06     56.20    17.39                             main
  0.02     56.21     0.01        1     0.01     6.48  f
  0.00     56.21     0.00        1     0.00    32.34  g
```

Each line describes one function

- **name**: name of the function
- **%time**: percentage of time spent executing this function
- **cumulative seconds**: [skipping, as this isn't all that useful]
- **self seconds**: time spent executing this function
- **calls**: number of times function was called (excluding recursive)
- **self s/call**: average time per execution (excluding descendents)
- **total s/call**: average time per execution (including descendents)

Number of seconds accounted for this function alone

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 69.34     38.81    38.81    60000     0.00     0.00  doit
 31.06     56.20    17.39                             main
  0.02     56.21     0.01        1     0.01     6.48  f
  0.00     56.21     0.00        1     0.00    32.34  g
```

Each line describes one function

- **name**: name of the function
- **%time**: percentage of time spent executing this function
- **cumulative seconds**: [skipping, as this isn't all that useful]
- **self seconds**: time spent executing this function
- **calls**: number of times function was called (excluding recursive)
- **self s/call**: average time per execution (excluding descendents)
- **total s/call**: average time per execution (including descendents)

Average number of sec per call
Spent in this function alone

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 69.34     38.81    38.81    60000     0.00     0.00  doit
 31.06     56.20    17.39                             main
  0.02     56.21     0.01        1     0.01     6.48  f
  0.00     56.21     0.00        1     0.00    32.34  g
```

Each line describes one function

- **name**: name of the function
- **%time**: percentage of time spent executing this function
- **cumulative seconds**: [skipping, as this isn't all that useful]
- **self seconds**: time spent executing this function
- **calls**: number of times function was called (excluding recursive)
- **self s/call**: average time per execution (excluding descendents)
- **total s/call**: average time per execution (including descendents)

Using gprof – call graph

```
granularity: each sample hit covers 2 byte(s) for 0.02% of 56.21 seconds

index % time    self  children    called           name
[1]     100.0   17.39   38.82                    main [1]
                 0.00   32.34       1/1              g [3]
                 0.01    6.47       1/1              f [4]
-----------------------------------------------
                 6.47    0.00   10000/60000          f [4]
                32.34    0.00   50000/60000          g [3]
[2]      69.1   38.81    0.00   60000              doit [2]
-----------------------------------------------
                 0.00   32.34       1/1              main [1]
[3]      57.5    0.00   32.34       1              g [3]
                32.34    0.00   50000/60000          doit [2]
-----------------------------------------------
                 0.01    6.47       1/1              main [1]
[4]      11.5    0.01    6.47       1              f [4]
                 6.47    0.00   10000/60000          doit [2]
-----------------------------------------------
```

Each line describes one function

- **name**: name of the function
- **%time**: percentage of time spent executing this function
- **children**: total amount of time in children
- **self**: time spent executing this function
- **called**: number of times function was called (excluding recursive)

Using gprof – call graph

```
granularity: each sample hit covers 2 byte(s) for 0.02% of 56.21 seconds

index % time    self  children    called       name
           00.0                                main [1]
                                   1/1
                                   1/1
-----------------------------------------------
                6.47    0.00   10000/60000     f [4]
               32.34    0.00   50000/60000     g [3]
[2]      69.1  38.81    0.00   60000           doit [2]
-----------------------------------------------
                0.00   32.34       1/1         main [1]
[3]      57.5   0.00   32.34       1           g [3]
               32.34    0.00   50000/60000     doit [2]
-----------------------------------------------
                0.01    6.47       1/1         main [1]
[4]      11.5   0.01    6.47       1           f [4]
                6.47               00/60000    doit [2]
-----------------------------------------------
```

Unique index of this function

Total time propagated into this function by its children

Number of times was called

Current Function: g( )

Percentage of the `total' time spent in this function and its children.

total amount of time spent in this function

- **name**: name of the function
- **%time**: percentage of time spent executing this function
- **children**: total amount of time in children
- **self**: time spent executing this function
- **called**: number of times function was called (excluding recursive)

Diagnostic Output from Programs

- So far, we have concentrated on unit testing of object-oriented software
- Unit testing involves understanding and using the documented interface to the software
- However, it can be very useful to inspect the internal operation of the software at finer grain (less abstraction)
  - "**Diagnostic output**" from a running program

Relevance:
What do we need for Diagnostic Output for?

- Tracing
- Timing
- Profiling
- Logging
- Error reporting

The goal in general:

debugging and optimization

Tools for Diagnostic Output

Standard output and standard error

Debuggers

Java/JUnit assertions

- Java logging framework
- Java profiling framework

We will look at logging today…

Logging

- **Logging** means:
  - *automatically recording* diagnostic output from a program
- Logging output can be useful during development and testing, but also in production code:
  - A web server could log IP address of incoming http requests
  - A mail server could log basic info about each email received
  - … **what else can you think of?**
- Logging using standard error output, or ordinary file output, can be done
- However it is better to make use of a logging framework, which provides lots of useful functionality

Logging Framework Functionality

- A logging framework or API (Application Programmer Interface) usually provides ways to:
  - Specify the origin of a log entry (which application, which class, which method, etc.)
  - Specify the content of the log entry
  - Specify the level of importance of the log entry
  - Control what importance levels are actually being logged
  - Control where the log entries are recorded
  - Analyze resulting log files

Logging Frameworks

- Logging frameworks exist for most application environments
- Several commercial and free frameworks are available for Java

We will concentrate on the framework in the **java.util.logging** package, distributed as part of J2SE since version 1.4.

Classes in java.util.logging

- Important classes in the java.util.logging package:
  - `Logger`
  - `Handler`
    - Subclasses: `ConsoleHandler`, `FileHandler`, `SocketHandler`
  - `Formatter`
    - Subclasses: `SimpleFormatter`, `XMLFormatter`
  - `Level`

java.util.logging.Level

- The **Level** class
  - contains public static named constants used to specify the importance level of log messages, and to control which log records are logged
- From highest importance to lowest:

```
Level.SEVERE
Level.WARNING
Level.INFO
Level.CONFIG
Level.FINE
Level.FINER
Level.FINEST
```

Sample Logs From a Unix Firewall

Feb 24 01:47:47	unbound	11545:0	info: 0.000000 0.000001 96
Feb 24 01:47:47	unbound	11545:0	info: lower(secs) upper(secs) recursions
Feb 24 01:47:47	unbound	11545:0	info: [25%]=0.0512428 median[50%]=0.100943 [75%]=0.230668
Feb 24 01:47:47	unbound	11545:0	info: histogram of recursion processing times
Feb 24 01:47:47	unbound	11545:0	info: average recursion processing time 0.244025 sec
Feb 24 01:47:47	unbound	11545:0	info: server stats for thread 1: requestlist max 12 avg 0.407341 exceeded 0 jostled 0
Feb 24 01:47:47	unbound	11545:0	info: server stats for thread 1: 6361 queries, 4965 answers from cache, 1396 recursions, 3045 prefetch, 0 rejected by ip ratelimiting

Feb 21 16:17:43	sshd	79333	Accepted keyboard-interactive/pam for admin from 10.0.1.225 port 52217 ssh2
Feb 21 16:17:39	sshd	79333	user admin login class [preauth]
Feb 21 16:17:39	sshd	79333	user admin login class [preauth]
Feb 21 16:16:54	php-fpm	60620	/index.php: Successful login for user 'admin' from: 10.0.1.225 (Local Database)
Feb 21 16:14:19	sshd	37363	Fssh_packet_write_wait: Connection from invalid user keith 10.0.1.225 port 52147: Permission denied [preauth]
Feb 21 16:14:19	sshd	37363	user NOUSER login class [preauth]
Feb 21 16:14:19	sshd	37363	user NOUSER login class [preauth]
Feb 21 16:14:19	sshguard	25987	Blocking "10.0.1.225/32" for 120 secs (3 attacks in 296 secs, after 1 abuses over 296 secs.)
Feb 21 16:14:19	sshguard	25987	Attack from "10.0.1.225" on service SSH with danger 10.
Feb 21 16:14:19	sshd	37363	Invalid user keith from 10.0.1.225 port 52147
Feb 21 16:09:36	php-fpm	2182	/index.php: Successful login for user 'admin' from: 10.0.1.225 (Local Database)
Feb 21 16:09:30	sshguard	25987	Attack from "10.0.1.225" on service unknown service with danger 10.
Feb 21 16:09:30	php-fpm	2182	/index.php: webConfigurator authentication error for user 'admin' from: 10.0.1.225
Feb 21 16:09:23	sshguard	25987	Attack from "10.0.1.225" on service unknown service with danger 10.
Feb 21 16:09:23	php-fpm	2182	/index.php: webConfigurator authentication error for user 'admin' from: 10.0.1.225

java.util.logging.Level

- Levels are used in two ways:
  - When logging a message, a Level must be specified for that message
  - Each Logger and Handler has a Level set for it; log messages with a Level less than that are ignored

- Two other named constants exist that can be used for the second purpose:

`Level.ALL`	log every message
`Level.OFF`	ignore every message

java.util.logging.Logger

To use J2SE logging

- Call the static factory method Logger.getLogger(String) to obtain a Logger object
- Pass as argument a String that will identify this Logger:
  - Usually, the package-qualified name of the application class
- Keep a static pointer to this Logger object so all methods in your application that want to log messages can access it
- Then call appropriate instance methods of that Logger object to perform logging…

Basic Logger usage example

```
import java.util.logging.Logger;
import java.util.logging.Level;
public class L1 {
  // Intialize a logger for this class
  protected static Logger logger = Logger.getLogger("L1");

  public static void main(String argv[])  {
    // Log a INFO tracing message
       logger.info("Entering main()");
       try{
      int j = 1 / 0;
    } catch (Exception ex){
      // Log the error
      logger.log(Level.SEVERE,"Problem",ex);
    }
       // Log a FINE tracing message
       logger.fine("Leaving L1.main()");
  }
}
```

=

```
logger.log(Level.FINE, "Leaving L1.main()");
```

Basic Logger logging methods

- The Logger class provides instance methods to log a simple String message at each of the log levels:

```
public void severe (String msg)
public void warning (String msg)
public void info (String msg)
public void config (String msg)
public void fine (String msg)
public void finer (String msg)
public void finest (String msg)
```

- Many other logging methods exist see the javadoc
- Note that a message will not be logged if the Logger's level, or its Handler's level, is higher than the message's level

java.util.logging.Handler

- Each Logger object must have one or more Handler objects associated with it, to actually log any log messages
- Existing Handler classes:
  - **`ConsoleHandler`** : log to stderr. Each Logger has this handler by default.
  - **`FileHandler`** : log to a file, with controllable log file rotation.
  - **`SocketHandler`** : connect to a logging server over a TCP/IP socket.

java.util.logging.Formatter

- Each Handler object must have a Formatter object which it uses to format logging messages.
- Existing Formatter classes:
  - **`SimpleFormatter`** : the default for ConsoleHandler.
  - **`XMLFormatter`** : the default for FileHandler and SocketHandler.

Controlling logging

- The Logger class has methods which permit setting the minimum Level, adding and removing Handlers, etc.
- The Handler classes have methods which permit setting the minimum Level, setting a Formatter, etc.
- You can use those methods in your program, but then if you want to change logging Level for example, you have to edit your program and recompile
  - **Easier and more powerful is using a properties file**
- Then run your program telling it to read from a changed properties file; recompilation not necessary

Logging properties file

- See the javadoc online documentation for the format of a logging properties file

- If the properties file is named for example myprop, and your application is named MyApp, then launch it as follows:

```
java –Djava.util.logging.config.file=myprop MyApp
```

- That tells the application to read logging configuration from the myprop properties file. No need to edit or recompile the application

Next Lecture

1. Software Correctness and Efficiency
2. Cost of logging