Profiling of Algorithms

- *Profiling* refers to the experimental measurement of the performance of algorithms.

- Profiling techniques fall into two main categories:

  - *Instruction counting* – the number of times which particular instruction(s) are executed is measured.

  - *Clock-based timing* – the time required for certain blocks of code to execute is measured.

Each has advantages and disadvantages, which shall briefly be discussed.

The Augment-Run-Analyze Process

Regardless of the technique employed, all profiling techniques involve three distinct steps.

1. *Augmentation* – Special code is added to the original program. This purpose of this code is to generate data on the execution of the program.

2. *Execution* – The augmented program is run. The augmentation code generates timing data which are written to a special file.

3. *Analysis* – A special program, called the *analyzer*, is run on the timing data to generate a report on the performance of the program.

Instruction Counting

The technique of instruction counting is very simple; the number of times certain critical instructions are executed is recorded.

This is illustrated below for a simple bubble sort program.

```
begin
     comp_count ← 0;
     assign_count ← 0;
     for i ← 2 to array_size do
       for j ← array_size downto i do
         comp_count ← comp_count + 1;
         if a[j-1] > a[j]
           then {swap}
             temp ← a[j-1];
             a[j-1] ← a[j];
             a[j] ← temp;
             assign_count ← assign_count + 3;
          end if;
        end for;
      end for;
end program.
```

This illustration shows the augmentation of the original program.

Sometimes, it is necessary to run a series of test cases, as illustrated by the following example.

```
begin
  for k ← 1 to no_test_cases do
    local_comp_count ← 0;
    local_assign_count ← 0;
    a ← test_data[k]; {array assignment}
    for i ← 2 to array_size do
      for j ← array_size downto i do
        local_comp_count ← local_comp_count +1;
        if a[j-1] > a[j]
          then {swap}
            temp ← a[j-1];
            a[j-1] ← a[j];
            a[j] ← temp;
            local_assign_count
                    ← local_assign_count + 3;
        end if;
      end for;
    end for;
    comp_count[k] ← local_comp_count;
    assign_count[k] ← local_assign_count;
  end for;
end program.
```

Again, only the augmented program is shown. Analysis is a separate step.

Counters may also be used to record the number of calls to procedures.

```
var merge_count, mergesort_count: integer;
merge_count ← 0;
mergesort_count ← 0;

procedure mergesort
          (a: int_array; low, high: array_index)
   begin
     if low < high then
       mid ← (low + high) div 2;
       mergesort(a, low, mid);
       mergesort(a, mid+1, high);
       merge(low, mid, high, a);
     end if;
     mergesort_count ← mergesort_count + 1;
end procedure mergesort;

procedure merge (low, mid, high: array_index)
   var b: array[low, high]; {local array}
   begin
      p1 ← low; p2 ← mid + 1; p ← low;
      while p1 ≤ mid and p2 ≤ high do
        if a[p1] ≤ a[p2] then
          b[p] ← a[p1]; p1 ← p1 + 1;
        else
          b[p] ← a[p2]; p2 ← p2 + 1;
        end if;
        p ← p + 1;
      end while;
      if p1 ≤ mid then
        a[p..high] ← a[p1..mid];
      end if;
      a[low..p-1] ← b[low..p-1];
      merge_count ← merge_count + 1;
end procedure merge;
```

Advantages of instruction counting:

- It is extremely simple to implement.
- The measurement code does not introduce error into the quantities being measured.

Disadvantages of instruction counting:

- Instruction counts are not always definitive in measuring the “real” performance of an algorithm, or in identifying bottlenecks in their performance.

Actual systems which used instruction counting for profiling:

- In early versions of Berkeley UNIX, (early to mid 1980’s) there was a Pascal interpreter called px. Associated with it was a counting profiler called pxp.

Clock-Based Timing

There are two principal flavors of clock-based timing of algorithms.

- Fixed-position logging.
- Random-sample logging.

Each approach has its advantages and disadvantages.

Fixed-Position Logging

The idea is to plant, within the program to be profiled, instructions which will log the elapsed running time. This is best illustrated via an example.

```
begin
     begin_time ← clock();
     for i ← 2 to array_size do
       for j ← array_size downto i do
         if a[j-1] > a[j]
           then {swap}
             temp ← a[j-1];
             a[j-1] ← a[j];
             a[j] ← temp;
         end if;
       end for;
     end for;
     end_time ← clock();
     elapsed_time ← end_time – begin_time;
end program.
```

Basic contraints:

- The function `clock()` must measure the amount of time which has been allocated to the program which is being profiled.
- A “time-of-day” clock, or “system-uptime” clock, is not appropriate.
- UNIX provides access to such a clock via the `getitimer` calls.

Clock accuracy and granularity:

- A digital clock “ticks” at discrete intervals.
- The length of this interval is called the *granularity* of the clock.
- The internal clock on a modern computer has a very small granularity – less than a nanosecond.
- However, the clocks which are accessible via system calls often has much higher granularities – on the order of hundredths of a second. (Typical examples are 10 ms. and 1/60 sec.)

Note: The *accuracy* of the clock refers to how close the length of the ticks are to the advertised interval. Clocks on modern digital computers are crystal controlled, and are extremely accurate. Unfortunately, the textbook confuses these two concepts, and uses the term “accuracy” to denote granularity.

- The granularity must be considered when constructing a profiling experiment. For example, the entire program above could very well run without the profiling clock ticking at all, in which case it would appear to run in zero time.

One solution is to run the program many times, and then average the results.

```
begin
  begin_time ← clock();
  for k ← 1 to no_repeats do
    a ← initial_array_data;
    for i ← 2 to array_size do
      for j ← array_size downto i do
        if a[j-1] > a[j]
          then {swap}
            temp ← a[j-1];
            a[j-1] ← a[j];
            a[j] ← temp;
        end if;
       end for;
     end for;
  end for;
  end_time ← clock();
  run_time
      ← (end_time – begin_time) / no_repeats;
end program.
```

The value of `no_repeats` should be chosen so that the difference `end_time – begin_time` is much greater than the clock granularity.

Consider the following alternative:

```
begin
  for k ← 1 to no_repeats do
    a ← initial_array_data;
    begin_time[i] ← clock();
    for i ← 2 to array_size do
      for j ← array_size downto i do
        if a[j-1] > a[j]
          then {swap}
            temp ← a[j-1];
            a[j-1] ← a[j];
            a[j] ← temp;
        end if;
      end for;
    end for;
    end_time[i] ← clock();
  end for;
  run_time ← 0;
  for k ← 1 to no_repeats do
    run_time ← run_time +
     (end_time[k] - begin_time[k]);
  end for;
  run_time ← run_time / no_repeats;
end program.
```

Disadvantage over previous program:

- The many time-recording statements introduce noise and inaccuracy into the final measurement.

Advantage:

- The time to re-initialize the array is not measured.

Consider the following example:

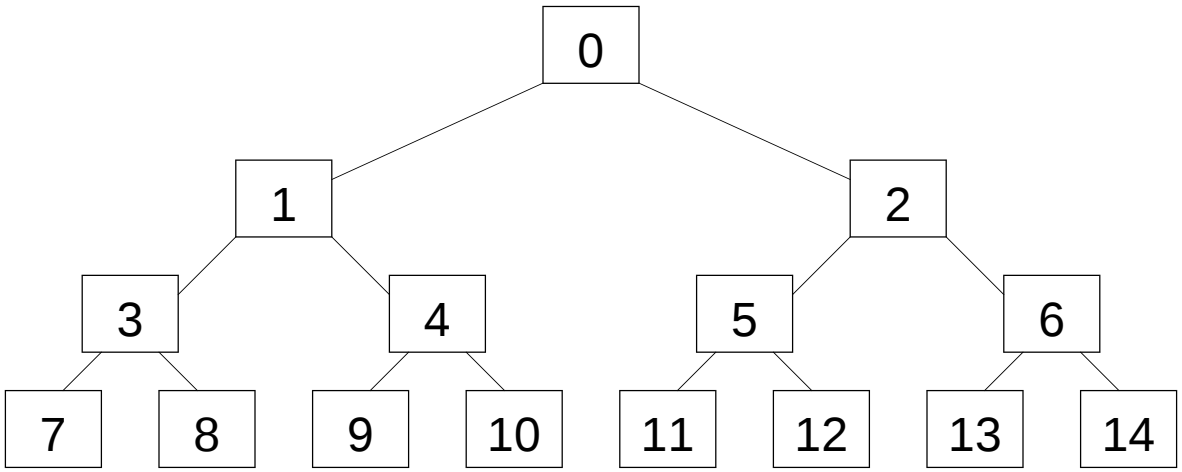
```
procedure mergesort
          (a: int_array; low, high: array_index)
   begin
     write_time_marker
                  (“begin”,”mergesort”,clock());
     if low <  high then
       mid ← (low + high) div 2;
       mergesort(a, low, mid);
       mergesort(a, mid+1, high);
       merge(low, high, mid);
     end if;
     write_time_marker
                  (“end”,”mergesort”,clock());
end procedure mergesort;

procedure merge (low, mid, high: array_index)
   var b: array[low, high]; {local array}
   begin
      write_time_marker
                  (“begin”,”merge”,clock());

      p1 ← low; p2 ← mid + 1; p ← low;
      while p1 ≤ mid and p2 ≤ high do
        if a[p1] ≤ a[p2] then
          b[p] ← a[p1]; p1 ← p1 + 1;
        else
          b[p] ← a[p2]; p2 ← p2 + 1;
        end if;
        p ← p + 1;
      end while;
      if p1 ≤ mid then
        a[p..high] ← a[p1..mid];
      end if;
      a[low..p-1] ← b[low..p-1];
      write_time_marker
                  (“end”,”merge”,clock());
end procedure merge;
```

The procedure `write_time_marker` places data into a file, which are later processed. A typical data file is shown on the next slide.

The times are all shown as just “t_”. For each time marker, the corresponding node in the call graph is shown in brackets. The numbers of the nodes of the call graph is shown below.

```
begin mergesort t_ [0]
begin mergesort t_ [1]
begin mergesort t_ [3]
begin mergesort t_ [7]
end   mergesort t_ [7]
begin mergesort t_ [8]
end   mergesort t_ [8]
begin merge     t_ [3]
end   merge     t_ [3]
end   mergesort t_ [3]
begin mergesort t_ [4]
begin mergesort t_ [9]
end   mergesort t_ [9]
begin mergesort t_ [10]
end   mergesort t_ [10]
begin merge     t_ [4]
end   merge     t_ [4]
end   mergesort t_ [4]
begin merge     t_ [1]
end   merge     t_ [1]
end   mergesort t_ [1]
begin mergesort t_ [2]
```

```
begin mergesort t_ [5]
begin mergesort t_ [11]
end   mergesort t_ [11]
begin mergesort t_ [12]
end   mergesort t_ [12]
begin merge     t_ [5]
end   merge     t_ [5]
end   mergesort t_ [5]
begin mergesort t_ [6]
begin mergesort t_ [13]
end   mergesort t_ [13]
begin mergesort t_ [14]
end   mergesort t_ [14]
begin merge     t_ [6]
end   merge     t_ [6]
end   mergesort t_ [6]
begin merge     t_ [2]
end   merge     t_ [2]
end   mergesort t_ [2]
begin merge     t_ [0]
end   merge     t_ [0]
end   mergesort t_ [0]
```



- In a “professional” package, software tools to perform automatic augmentation, as well as analysis, are available.

- Since automatic augmentation is a relatively complex process, it is common in “roll-your-own” applications to perform manual augmentation, but to write a program to perform analysis.

Systems which perform fixed-position logging:

- The University of Minnesota Pascal profiling system for the CDC Cyber systems (1970’s).

- Software assignment 1.

Advantages of fixed-position logging:

- Very detailed information about performance may be obtained.

- Caller-specific information about performance may be obtained. For example, if procedure C is called by both procedure A and procedure B, separate information about the performance of C under each caller may be obtained.

- It is not necessary to modify the compiler in any way. The existing compiler can be used.

Disadvantages of fixed-position logging:

- The timing instructions themselves consume time, and so introduce errors into the measurement process.

- Special measures must be taken to make sure that enough measurements are taken to avoid problems with clock granularity.

- It is essential to have available a clock which can measure the amount of time which has been allocated to the program being profiled. (Operating system support)

Random-Sample Logging

To overcome some of the problems encountered in fixed-position logging, random-sample logging is sometimes used.

Idea:

- At random times, sample the execution of the program, determining which procedure/statement is executing.
- From these samples and knowledge of the total running time of the program, construct a profile of how much time is spent executing each procedure/statement.

Implementation:

- Generally, this approach requires compiler support.

- At compilation, a table which matches addresses to routines is constructed.

- During execution, the sampler looks at the address of the executed instruction, and from that determines which instruction is executing. These data are recorded in a file.

- The absolute number of times which each routine is called is also computed via special code which is constructed during compilation.

- Total running time may be determined from the sample rate and the number of samples collected. No other measurement of total running time is necessary.

- As in fixed position logging, there is a separate analyze phase during which the data file is crunched, and program statistics generated. From the number of times which each procedure was invoked, together with the statistical distribution of how often the program was found executing a given procedure, relatively informative profiles may be generated.

Example:

- Sample rate =1000/second.
- 10000 samples taken.
- Recorded call data:

Procedure	No. Calls	No. Sample Hits
A	300	1000
B	600	2000
C	1000	7000

We may then compute the following:

- Total running time = 10 seconds.

Procedure	Total time	Time per call
A	1 sec.	3.33 msec.
B	2 sec.	3.33 msec.
C	7 sec.	7.00 msec.

Advantages of random-sample logging:

- Clock granularity is not an issue.
- There are fewer timing instructions to generate noise and corrupt the measurements.

Disadvantages of random-sample logging:

- Compiler support is required. It is difficult to “roll your own.”
- Call sequences are difficult to determine. (If procedure C is called by both procedure A and procedure B, it is difficult to determine how much of the execution time of C is attributable to each type of call.)

Systems which perform random-sample logging:

- The `gprof` profiling system under UNIX.