Performance*

• Objective: To learn when and how to optimize the performance of a program.

• “The first principle of optimization is don’t.”
  – Knowing how a program will be used and the environment it runs in, is there any benefit to making it faster?

*The examples in these slides come from Brian W. Kernighan and Rob Pike, “The Practice of Programming”, Addison-Wesley, 1999.
Approach

- The best strategy is to use the simplest, cleanest algorithms and data appropriate for the task.
- Then measure performance to see if changes are needed.
- Enable compiler options to generate the fastest possible code.
- Assess what changes to the program will have the most effect (profile the code).
- Make changes one at a time and re-assess (always retest to verify correctness).
  - Consider alternative algorithms
  - Tune the code
  - Consider a lower level language (just for time sensitive components)
Topics

• A Bottleneck
• Timing and Profiling
  – time and clock
  – algorithm analysis
  – prof and gprof
  – gcov
• Concentrate on hot spots
• Strategies for speed
• Tuning the code
A Bottleneck

• isspam example from the text
  – Heavily used
  – Existing implementation not fast enough in current environment
    • Benchmark
    • Profile
  – Tune code
  – Change algorithm
Timing

• In Unix environment
  – time command

• Example (time quicksort from chapter 2)
  – head –10000 < /usr/share/dict/words | shuffle > in
  – gcc –o sort1 sort1.c quicksort.c
  – time sort1 < in > /dev/null
Algorithm Analysis

- Consider the asymptotic analysis of your program and the algorithms you are using
- For quicksort, let $T(n)$ be the runtime as a function of the size of the input array (the time will depend on the particular input array!)
- The expected runtime is $\Theta(n\log n)$
  - If each partition roughly splits the array in half then the computing time $T(n) \approx 2T(n/2) + cn$
- The worst case is $\Theta(n^2)$
  - If each partition splits the array into two pieces of unequal size (in the extreme 1 and $n-1$)
  - $T(n) = T(n-1) + cn = \Theta(n^2)$
Solving Recurrences By Repeated Substitution

- \( T(n) = 2T(n/2) + cn \)
  
  \[
  = 2[2T(n/4) + c(n/2)] + cn \\
  = 4T(n/4) + cn + cn = 2^2T(n/2^2) + 2cn \\
  = 2^2[2T(n/2^3) + c(n 2^2) ] + 2cn \\
  = 2^3T(n/2^3) + 3cn \\
  
  \]

\[
= 2^jT(n/2^j) + jcn \\
\]

Suppose \( n = 2^k \) and \( T(1) = c \) \( \Rightarrow \) \( T(n) = 2^kT(1) + kcn \)
\( = n + cn\log(n) \)
Solving Recurrences By Repeated Substitution

- \( T(n) = T(n-1) + cn \)
  \[= T(n-2) + c(n-1) + cn \]
  \[= T(n-3) + c(n-2) + c(n-1) + cn \]
  \[\ldots \]
  \[= T(n-j) + c[(n-j+1) + \ldots + n)] \]

Suppose \( T(1) = c \Rightarrow \)

\[T(n) = T(n-(n-1)) + c[(n-(n-1)+1 + \ldots + n] \]
\[= T(1) + c[2+3+\ldots+n] = c[1+2+\ldots+n] \]
\[= c \sum_{j=1}^{n} j = cn(n+1) / 2 \]
Worst Case for Quicksort

• Modify the code to remove the random selection of the pivot
  – This makes it possible to deterministically construct a worst case input (this is why randomization was used)
  – The worst case will occur for sorted or reverse sorted input
  – For sorted input, the number of comparisons \( Q(n) \) as a function of input size satisfies
    – \( Q(n) = Q(n-1) + n-1, \) \( Q(1) = 0 \)
    – \( Q(n) = n(n-1)/2 \)
What does Asymptotic Analysis mean for Actual Runtimes

• If $T(n) = \Theta(n^2)$
  – Doubling the input size increases the time by a factor of 4
  – $T(2n)/T(n) = (c4n^2 + o(n^2))/(cn^2 + o(n^2))$, which in the limit is equal to 4. $o(n^2)$ means lower order terms.

• If $T(n) = \Theta(n \log n)$
  – Doubling the input size roughly doubles the time [same as linear]
  – $T(2n)/T(n) = (c2n \log(2n) + o(n \log n))/(n \log(n) + o(n \log n)) = 
    = (c2n \log n + o(n \log n))/(c n \log n + o(n \log n))$, which in the limit is equal to 2
Empirical Confirmation

- Run and time quicksort (without random pivot) on sorted inputs of size 10,000 and 20,000, and 40,000
- Compute the ratio of times to see if it is a factor of 4.

- What if random inputs are used?
Growth Rates and Limits

• Suppose $T(n) = \Theta(f(n))$ [grows at same rate]
  – $\lim_{n \to \infty} T(n)/f(n) = c$, a constant $> 0$.
  – [Actually this is not true, there may be separate limsup and liminf, but as a first approximation you can view it is true.

• Suppose $T(n) = o(f(n))$ [grows slower]
  – $\lim_{n \to \infty} T(n)/f(n) = 0$

• Suppose $T(n) = \omega(f(n))$ [grows faster]
  – $\lim_{n \to \infty} T(n)/f(n) = \infty$
Determining Growth Rate Empirically

• Time quicksort with a range of input sizes
  – e.g. 1000, 2000, 3000, …, 10000

• Write a program that times sort for a range of inputs. Use the clock function to time code inside a program.
  – T(1000), T(2000), T(3000),…,T(10000)
  – plot times for range of input to visualize
  – Compute ratios to compare to known functions
  – T(1000)/1000², T(2000)/2000²,…, T(10000)/10000²
  – Does the ratio approach a constant, go to 0, go to ∞?
  – i.e. is is growing at the same rate, faster, or slower than the comparison function?
Obtaining Range of Times

• sortr 1000 10 1000
  – sorts and times sorted arrays of size
  – 1000, 2000, 3000,…,10000
Profiling with gprof

- Reports on time spent in different functions (also gives number of times functions called)
  - Shows the hotspots

- `gcc -pg sort1.c quicksort.c -o sort1`
- `sort1 < in.40000 > /dev/null`
- `gprof sort1 gmon.out`
Profiling with gcov

- Uses source code analysis provided by the compiler to analyze the number of times each statement in the source code is executed.

  - `$gcc -fprofile-arcs -ftest-coverage sorti.c quicksorti.c -o sorti`
  - `$sorti 10`
  - `$gcov sorti.c`
  - `$gcov quicksorti.c`