Code Complete, Chapter 25, Code-Tuning Strategies

Improving performance
- check requirements (does it really have to be that fast?)
- program design (interactions between components)
- class and routine design (choosing good data structures & algorithms)
- faster hardware
- more efficient language (interpreted languages are slow)
  - table on pg. 600 (pg. 15 in PDF)
- better compiler (turn on compiler's optimizations)
- operating system interactions (too many system calls, which are slow)
- remove programming errors
  - passing large objects by value
  - leaving debugging code turned on (tracing, logging, etc.)
  - leaving asserts turned on
- memory hierarchy
  - registers
  - cpu cache
  - ram
  - disk (virtual memory)
  - remote computer

  - avoid unnecessary I/O operations
    - bring data into memory and operate on it there instead of manipulating it on disk or across a network
  - locality of reference to avoid cache misses and virtual memory paging
    - example on pg. 599 (pg. 14 in PDF)
    - row-major order more efficient than column major order
- code tuning (not the most effective)

code optimizations usually make a program harder to read
- readability is very important
- don't optimize until you know where program is spending its time
- program spends 80% of its time in 20% of its code (Boehm)
- 4% of the code accounts for 50% of the execution time (Knuth)
- on the other hand, you should avoid doing things that are unnecessarily inefficient (use good programming practices)

modular design allows only the critical parts to be optimized or replaced without affecting other code

1. Good design
2. Make it work
3. Profile
4. Optimize critical pieces

always measure the effect of an optimization
- profilers
- optimization may actually harm performance

```
Run profiler on Web Cloner on rodham-server
-pg flag to compile AND link
run program to generate gmon.out
run gprof to print reports
      gprof –p bin/cloner gmon.out  # print flat profile
      gprof –q bin/cloner gmon.out  # print call graph
```

Code Complete, Chapter 26, Code-Tuning Techniques

Logic

- Stop when you know the answer
  - example on pg. 611 (pg. 3 in PDF)

- Order tests by frequency
  - example on pg. 612 (pg. 4/5 in PDF)

- Compare performance of similar logic structures
  - switch vs. cascaded if-else
  - timing table on pg. 614 (pg. 6 in PDF)

- Substitute table lookups for complicated expressions
  - example on pg. 615 (pg. 7 in PDF)

- Use lazy evaluation
  - Avoid doing work until the result of the work is needed
  - If the result is never needed, the work is never done
  - Example:
  - A program uses a large table of values, but only a small fraction of the values are used in any given run.
  - Rather than computing the entire table up front, just compute the entries that are actually needed dynamically

Loops

Unswitching
example on pg. 616 (pg. 9 in PDF)

Combining Loops
example on pg. 617/618 (pg. 10 in PDF)

Unrolling
example on pg. 618/619 (pg. 11 in PDF)
single and double unroll

Minimizing the work inside loops
example on pg. 620 (pg. 13 in PDF)

Sentinel Values
example on pg. 621/622 (pg. 14/15 in PDF)

Putting the busiest loop on the inside
example on pg. 623 (pg. 16 in PDF)

Before

	lcv init's	cond check	lcv increments
outer	1	100	100
inner	100	500	500

After

	lcv init's	cond check	lcv increments
outer	1	5	5
inner	5	500	500

Replace multiplications that depend on the loop index with addition
example on pg. 624 (pg. 17 in PDF)

Data Transformations

- Use integers rather than floating-point numbers
  - example on pg. 625 (pg. 18 in PDF)

- Use the fewest array dimensions possible
  - example on pg. 625/626 (pg. 19 in PDF)

- Minimize array references
  - array references take time (remember Manual Indexing for 2D arrays)
  - example on pg. 626/627 (pg. 20 in PDF)

- Store sizes of variable-length data structures rather than computing on demand
  - strings store length variable (as opposed to C-strings where length must be computed)
  - data structures such as BSTs and hash tables store current number of elements

- Sort key indexes rather than complete objects
  - when sorting arrays of large objects, moving large objects around is expensive, instead create an index array that contains (key, obj ref), and sort the index array instead
  - this technique may allow an in-memory sort when objects are too large to store in memory

- Use caching
  - store previously computed results and reuse when possible
  - example on pg. 628/629 (pg. 21/22 in PDF)

Expressions

Exploit algebraic identities

- not A and not B [3 operations]
- not (A or B) [2 operations]
- sqrt(x) < sqrt(y) <==> x < y
- timing table on pg. 630 (pg. 23 in PDF)

Use strength reduction

- replace an expensive operation with a cheaper one
- show list of strength reductions on pg. 630 (pg. 24 in PDF)

Initialize at compile time

- example on pg. 632 (pg. 26 in PDF)

Be wary of system routines

- system routines provide lots of precision
- this makes them slow
- often the precision is unnecessary, and we can write less precise routines that are much faster
- example on pg. 633/634 (pg. 26/27 in PDF)

Use the correct type of constants

- example on pg. 635 (pg. 28 in PDF)

Precompute results

- example on pg. 636/637 (pg. 29/30 in PDF)

Eliminate common subexpressions

- example on pg. 638/639 (pg. 32 in PDF)

Routines

Inline routines

- inline functions in C++

Recode in a lower-level language

- Java => C/C++
- C/C++ => Assembly
- example on pg. 641/642 (pg. 35/36 in PDF)