Chapter 26: Code Tuning Techniques

Logic

Stop Testing When You Know the Answer

- If (5 < x) and (x < 10)
    - Don't need to do second half of test
- Stop iterating through things once you find the answer

Order Tests by Frequency

- Arrange tests so the one that's fastest and most likely to be true is performed first

Compare Performance of Similar Logic Structures

- Like comparing if-then-else to case
- For to while

Substitute Table Lookups for Complicated Expressions

Use Lazy Evaluation

- Only do exactly what needs to be done at the last possible moment

Loops

Unswitching

- Switching refers to making a decision inside a loop every times its executed
- Get rid of this lmao
    - Make the decision outside of the loop

Jamming

- Two loops that operate on the same data
    - Make operations happen in the same loop so only have to iterate once

Unrolling

- Like instead of looping just access operations directly in line
- Python times actually decrease for this so don't worry about it

Minimizing the Work Inside Loops

- If you can evaluate part of a statement outside of the loop, do it

Putting the Busiest Loop on the Inside

- When you have nested loops, think about which one you want on the inside and which you want on the outside
- Inner loop should execute more often than the outer loop

Strength Reduction

- Means replacing an expensive operation with a cheaper one
    - Multiplication for addition

Data Transformations

Use Integers Rather Than Floating Point Numbers

Use the Fewest Array Dimensions Possible

Minimize Array References

Use Supplementary Indexes

- Means adding related data that makes accessing a data type more efficient
- Can add the related data to the main data type, or store in parallel
- Mainly reduce repeated operations

Use Caching

Expressions

Exploit Algebraic Identities

- You can use identities to replace costly operations with cheaper ones
    - Not a and not b
    - → not (a or b)

Use strength reduction

- Replace multiplication with addition
- Replace exponentiation with multiplication
- Replace trig routines with trig identities
- Replace floats to fixed place or ints
- Replace multiplication and division with shift operations

Initialize at Compile Time

Be Wary of System Routines

Use the Correct Type of Constants

Precompute Results

- Compute on the fly or compute them once, save them, and look them up as needed
- This can take several forms:
    - Computing results before the program executes, and wiring them into constants that are assigned at compile time

    - Computing results before the program executes, and hard coding them into variables used at run time
    - Computing results before the program executes, and putting them into a file that's loaded at run time
    - Computing results once, at program startup, and then referencing them each time theyre needed
    - Computing as much as possible before a loop begins, minimizing the work done inside the loop
    - Computing results the first time theyre needed, and storing them so that you can retrieve them when theyre needed again

Eliminate Common Subexpressions

- Instead of recalculating value/45, just set that to a variable

Routines

- Small well-defined routines save spaced because they take the place of doing jobs separately in multiple places

Rewrite Routines Inline

- Sometimes a little slower for a routine to not be inline

Recoding in a Low-Level Language

- A long-standing piece of advice says if you run into a performance bottleneck, you should recode in a low level language
- Typical approach is:
    - Write 100% of an application in a high level language
    - Fully test the application and verify its correct
    - If performance improvements are not needed after then, profile the hotspots
    - Recode a few small pieces in a low level language to improve performance