Code optimization is often a tradeoff between

- Execution time (“speed”)
- Memory usage
- Code (executable) size
- Maintainability/readability

Rules of thumb:

- Using the right data structure or algorithm is much more important than micro-optimizations
- Make the common case fast... don't worry about code that runs infrequently (80/20 rule)
- Don't guess! Measure!
- don't do unnecessary work
- don't unnecessarily allocate memory

**Code profiling**

We want to know...

- where it spends its time
- how many times functions are called
- how much memory it uses
- how long it takes to run (CPU time, number of instructions, etc.)

**GNU gprof: code profiler for C**

```bash
cc -g -pg -o myprogram myprogram.c
./myprogram (produces gmon.out)
gprof myprogram gmon.out > gprof.out
```

The output file indicates:

- % of time spent in each function
- # of seconds spent in that code
- # of times the function is called
- ms/call per function

you can ignore “mcount” and “profil”; those are part of gprof

**Massif: memory (heap) profiler for C**

```bash
cc -g -o myprogram myprogram.c
valgrind --tool=massif --time-unit=B ./myprogram
ms_print massif.out.[pid]
```

Graph shows heap usage over time
Table shows heap usage at each snapshot

- useful: actually being used
- extra: allocated but not used (admin bytes in block, padding, etc.)

Detailed snapshot shows trees of function calls and what % of heap each is responsible for

To see the assembly language code that gets generated: `cc -S foo.c`
C Compiler Optimizations
The C compiler is smart, and makes some basic optimizations that you do not have to worry about:

Inlining: replace simple function calls with the code itself, so as not to jump unnecessarily; you can use the __inline descriptor to force this (what's the tradeoff? Increase code size!)

Dead Code Elimination: remove unreachable code

Constant Folding: combine literals

Strength Reduction: use simpler arithmetic operations

Loop Unrolling: call loop body directly (numerous times) if number of loops can be determined and is small (tradeoff?)

Compiler Flags:
- With -O, the compiler tries to reduce code size and execution time, without performing any optimizations that take a great deal of compilation time.
- -O2 optimizes even more. GCC performs nearly all supported optimizations that do not involve a space-speed tradeoff. As compared to -O, this option increases both compilation time and the performance of the generated code. -O2 turns on all optimization flags specified by -O.
- -O3 even more stuff beyond -O2
- -Os enables all -O2 optimizations that do not typically increase code size. It also performs further optimizations designed to reduce code size.

Lazy Evaluation

Delay computation until you know you need the result
```c
int x = gigantic_function( );
if (y < 0) return;
else y = x + 8;
```

Do easy things first:
```c
if (easy() || hard())...
```

Save the result of the function call:
```c
for (i = 0; i < strlen(msg); i++)
```
In Lab Assignment #2, why was the LinkedList implementation so much slower than the ArrayList?

```java
LinkedList list0 = ... 
LinkedList list1 = ... 
for (int i = 0; i < list0.size(); i++)
    for (int j = 0; j < list1.size(); j++)
        if (list0.get(i).equals(list1.get(j))) ...
```

The reason is that list0.get(i) is called repeatedly in the inner loop, even though it doesn't need to be (since its value does not change in the inner loop). In a LinkedList, the get function is O(n), so this is unnecessarily taking a lot of time; it's O(1) in an ArrayList, of course.