

**Performance: do we need it?**

**Lecture 19**

**Basic performance tuning**

After we have an idea about the mechanisms that allows programs to be extended, we get 80% of the coding problem done.

The remaining 20% requires that our code is efficient. We will briefly examine a few things that you should look for when improving performance.

**References:**

- Programming Pearls Column 9 and 10, 2nd Ed. Jon Bentley, Addison Wesley.

There are a lot of virtues that we want our code to exhibit. Examples:

- **Correct:** it does the right thing.
- **Robust:** it won't cause a catastrophe on wrong input.
- **Simple and small:** it doesn't take 3 man-years to understand.
- **Maintainable:** modifying a subsystem doesn't need a complete rewrite.
- **Extensible:** it can be used in many ways.
- **Portable:** it can be used in different computers.
- **Efficient:** it makes good use of computational resources like memory and CPU cycles.

Any of them can affect reusability. Sometimes trade-offs has to be made. Efficiency is not particularly high in the list, especially since we usually have ample resources to play with.

PMOOP(0396A)

PMOOP(0396A)-19.1

**When to introduce efficiency?**

But it does not mean that we don't need to deal with efficiency.

- It might be okay for one operation to take 1 second in one application. But that means we cannot use the code interactively if 200 of them runs in a row.
- It might be okay for a program to take 128M RAM if we have 512M RAM anyway. But then don't expect 10 such program to be executed concurrently.

The following is usually a right advice:

**Make it correct first, make it efficient later.**

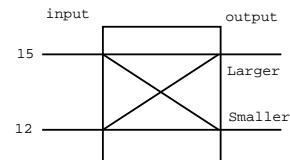
Corollary: the machine that you do development (especially early development) needs to be have much more resources than your target machine.

**Estimate running time**

Before working on a program, you should have an idea about what is feasible. Your problem might be too hard to solve at the first place, and any effort to program it will be wasted.

**Example:** in ACM'01 programming contest world final we see the following question (simplified here):

We are given some hardware pieces (comparators) that accepts two numbers in its two input, and output the two input numbers in the correct order:

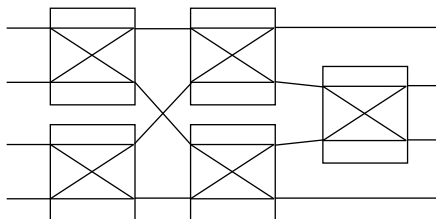


PMOOP(0396A)-19.2

PMOOP(0396A)-19.3

**Example (cont'd)**

The question is, given a network of these comparator, will it always sort the input? E.g.: 4 input sorter:



We know that the input contains at most 12 numbers, and there are at most 1000 comparators. We also know that the judges will probably think that the program is dead if it does not say anything in 1 minute. Should we do the question?

**An easy algorithm**

Our team actually tried the problem, with the following simple idea: it sorts correctly if it gives the same output given any permutation of the list (1..n), where n is the input size. The algorithm:

```

foreach permutation p of (1..n)
  put p into the input of the network
  calculate the output of each comparator
  if it is not sorted
    return false
return true

```

- Quick estimates:** Number of permutations =  
Amount of instruction for each loop =  
Amount of time per instruction =  
Amount of time needed =

**Verdict:**

PMOOP(0396A)-19.4

PMOOP(0396A)-19.5

### Finding performance hot-spots

So suppose you have done feasibility analysis, started to program and now it comes the time to improve performance. **Rule of thumb:**

**80% of the running time is usually spent on 20% of the code.**

This is both a good news and a bad news:

- **Good:** the remaining 80% of your code won't need top efficiency. Performance tuning should be done **rarely**.
- **Bad:** if you optimize performance in the wrong place, the pay-off is really slim. **You need to know whether it pays off before code tuning.**

Before performance tuning, **find which part of your program needs it.** This means we need a tool to **profile** the code. Then we can estimate the amount of speed-up you can expect by tuning each part, and determine whether we should spend our time there.

PMOOP(0396A)-19.6

### The tool in Unix

We need a tool that tell us the time spent in each function, the number of times the program runs there, etc.

For Unix code, we use **time** and **gprof** to perform profiling.

The **time** tool needs no special attention. It just measures how much time is used for the whole program, and shows how much of them is running within the OS. It also shows memory usage.

The **gprof** tool requires that you compile your program like this:

```
g++ -Wall -O2 -pg a.cc
```

This compile the program in such a way that special code will be generated to remember the amount of time spent in each function.

After you **run your program**, such information will be stored in a file called `gmon.out`. You can run **gprof** to read the data in it.

PMOOP(0396A)-19.7

### How to improve efficiency

Efficiency can be improved by many different methods:

- Use the **right data structure:** take the DSA course.
- Use the **right algorithm:** take the algorithms course.
- Use the **right programming language and tool.**
- Turn on **optimization, inline** right: we know all that.
- **Code tuning**, i.e., minor adjustments to make code faster: our focus here.

Again, code tuning is not particularly high in the list, meaning that it is not always applicable.

When you need a bit more efficiency, try to examine each, see which is the most promising and go for that. When you need a lot more efficiency, try them all.

PMOOP(0396A)-19.8

### Rules of code tuning

What "minor adjustment" are possible? Here are some:

- **Trade memory for running-time.** Sometimes a lot of running time can be saved if we just store a bit more data.
- **Trade running-time for memory.** Sometimes you can save a lot of memory by slowing down a bit.
- **Make better loops.** Since loops are often executed many times, saving a few cycles in each of them gives you quite some performance.
- **Get better knowledge about the specific area.** Sometimes a naive solution simply computes far more than what is needed.
- **Try to be nice to the processor cache.** The CPU reduces the amount of memory fetches by using a small cache. Avoid overflowing it.
- **Reuse results.** Avoid computing the same thing twice.

PMOOP(0396A)-19.9

### Trading memory for running time

**Example:** what to do if you frequently need to spread off the bits of a number, e.g., from binary 10100100 to 1100110000110000?

You can calculate it:

You can also hard-code it:

```
x = input;
y = 0;
for (int i=0; i<8; ++i) {
    y = y >> 2;
    if (x & 1)
        y |= 0xc000;
    x = x >> 1;
}
return y;
```

```
static short result[] = {
    0x0, 0x3, 0xc, 0xf,
    0x30, 0x33, 0x3c, 0x3f,
    0xc0, 0xc3, 0xcc, 0xcf,
    0xf0, 0xf3, 0xfc, 0xff,
    0x300, 0x303, 0x30c, 0x30f,
    ... // 59 more lines
};
return result[input];
```

With only 512 bytes of memory, we get more than 8 times the speed of the original code. So the memory pays off very nicely.

PMOOP(0396A)-19.10

### Trading running time for memory

**Example:** Suppose we want to compute all primes up to some big number, say 10000000. One method is to allocate an array of that size, and start crossing out every multiple of 2.

Then we know that 3 is not crossed out, so 3 is a prime. We then go cross out every multiple of 3. We find that 5 is not crossed out. Continuing this way we can find every prime.

But what to do if 10M RAM is too much for us? One thing we can do: squeeze 8 bits into one char to reduce the memory requirement 8-fold.

This requires a bit more time for accessing individual elements, so the program may be slower by a factor of 2, but reducing memory requirement is more important for us.

*Note: if you actually write the two program and try it, you'll find that the program runs faster, instead of slower, since you are much nicer to cache this way. Speed-up: it may be as much as 2.)*

PMOOP(0396A)-19.11

### Dealing with loops

There are three different things that you can do to loops:

- **Take the common expression within the loop to outside**, so that it don't need to be recalculated.
- **Rearrange the loop** so that some expressions of the previous iteration can be re-used in the current iteration. Leave enough traits of the previous iteration to ease the computation of the current iteration.
- **Unroll the loop** to get more opportunity of optimization, reduce the number of loop condition checking, and reduce the number of jumps that the program need to perform.

The first bullet is done by most optimizing compilers if the loop does not involve function calls.

The remaining two normally need to be done manually.

PMOOP(0396A)-19.12

### Loop unrolling

**Example:** Suppose our program do a lot of sequential search, and we would pay anything if that function can be made quicker by a factor of 2.

**The original function:**

```
int search(int[] arr, int n,
          int target) {
    for (int i = 0; i < n; ++i)
        if (arr[i] == target)
            return i;
    return -1;
}
```

The unrolled code is longer but faster: less jumps, fewer adds, etc.

**Unrolling code 8-fold:**

```
int search(int[] arr, int n, int target) {
    int i;
    for (i=0; i+8 <= n; ++i) {
        if (arr[i] == target) return i;
        ... // 6 more lines
        if (arr[i+7] == target) return i+7;
    }
    for (; i < n; ++i)
        if (arr[i] == target)
            return i;
    return -1;
}
```

PMOOP(0396A)-19.13

### Understand the problem domain better

The best speed-up usually comes from things that is domain-specific.

**Example:** Let's see how the problem we have at the beginning of this lecture can be solved.

If we know sorting networks better, we know that we really don't have to test all permutations. Instead, we just need to test for all possible sequences of 0's and 1's. (Reference: MIT text on Algorithms, sorting network.)

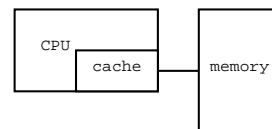
```
foreach sequence of n 0's of 1's
    put p into the input of the network
    calculate the output of each comparator
    if it is not sorted
        return false
return true
```

Since we reduced the number of iterations from 500M to 4k, the program is now 125000 times faster.

PMOOP(0396A)-19.14

### Be nice with cache

Cache is some fast memory in the CPU that mirror part of the memory:



However, to simplify the design of the CPU (and to make the rest of the computer fast), not all memory places can be placed in every cache entry.

In many computers, a **memory place has a fixed cache entry for it**, like hashing. The hash function is very simple: it takes the last few bits.

**Each time the CPU loads a line of cache**, e.g. 64 bytes.

This works well if you access things nearby. It works reasonably if you access things randomly. It crawls if you access every other 1024 bytes.

PMOOP(0396A)-19.15

### Be nice with cache, cont'd

Suppose you want to copy a matrix. Which function below is better?

```
float a[512][512];
float b[512][512];

for (int i = 0; i < 512; ++i)
    for (int j = 0; j < 512; ++j)
        a[i][j] = b[j][j];
```

```
float a[512][512];
float b[512][512];

for (int j = 0; j < 512; ++j)
    for (int i = 0; i < 512; ++i)
        a[i][j] = b[j][j];
```

The function on the left have **very good memory access pattern**: it access memory in a linear fashion. Most of the memory loads are from cache.

The function on the right **causes a cache miss everytime**, and all loads are from the memory.

The code on the left can be 6 times as fast as the code on the right.

PMOOP(0396A)-19.16

### Conclusion and Exercise

- Code tuning is something that should be done rarely, but pays back very nicely if it is done appropriately.
- Before code tuning, we should make sure that the performance is really needed, and code tuning is more effective than other performance boosters.
- Code tuning is usually very problem-specific, and the way to tune each piece of code is vastly different. But we have a few rules that we should look at for ideas.

**Exercise:**

- Actually write the bit-compressor suggested in the trading speed for memory example. Use grof to compare their speed.

PMOOP(0396A)-19.17