Strategy for developing efficient programs:

1. Design the program well
2. Implement the program well**
3. Test the program well
4. Only after you’re sure it’s working, measure performance
5. If (and only if) performance is inadequate, find the "hot spots"
6. Tune the code to fix these
7. Repeat measure-analyse-tune cycle until performance ok

(** see "Programming Pearls", "Practice of Programming", etc. etc.)

Rapid development of a prototype may be the best way to discover/assess performance issues.

Hence Fred Brooks maxim - "Plan To Throw One Away".
Typically programs spend most of their execution time in a small part of their code.

This is often quoted as the 90/10 rule (or 80/20 rule or ...):

“90% of the execution time is spent in 10% of the code”

This means that

- most of the code has little impact on overall performance
- small parts of the code account for most execution time

We should clearly concentrate efforts at improving execution speed in the 10% of code which accounts for most of the execution time.
Given the -p flag clang instruments a C program to collect profile information.

When the program executes this data is left in the file gmon.out.

The program gprof analyzes this data and produces:

- number of times each function was called
- % of total execution time spent in the function
- average execution time per call to that function
- execution time for this function and its children

Arranged in order from most expensive function down.

It also gives a *call graph*, a list for each function:

- which functions called this function
- which functions were called by this function
Program is slow on large inputs e.g.

$ clang -O3 word_frequency0.c -o word_frequency0
$ time word_frequency0 <WarAndPeace.txt>/dev/null

real 0m52.726s
user 0m52.643s
sys 0m0.020s
Performance Improvement Example - Word Count

We can instrument the program to collect profiling information and examine it with clang:

$ clang -p -g word_frequency0.c -o word_frequency0_profile
$ head -10000 WarAndPeace.txt | word_frequency0_profile >/dev/null
$ gprof word_frequency0_profile

.....

<table>
<thead>
<tr>
<th>% cumulative</th>
<th>self</th>
<th>self</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>time</td>
<td>seconds</td>
<td>seconds</td>
<td>calls</td>
</tr>
<tr>
<td>88.90</td>
<td>0.79</td>
<td>0.79</td>
<td>88335</td>
</tr>
<tr>
<td>7.88</td>
<td>0.86</td>
<td>0.07</td>
<td>7531</td>
</tr>
<tr>
<td>2.25</td>
<td>0.88</td>
<td>0.02</td>
<td>80805</td>
</tr>
<tr>
<td>1.13</td>
<td>0.89</td>
<td>0.01</td>
<td>1</td>
</tr>
<tr>
<td>0.00</td>
<td>0.89</td>
<td>0.00</td>
<td>2</td>
</tr>
<tr>
<td>0.00</td>
<td>0.89</td>
<td>0.00</td>
<td>1</td>
</tr>
<tr>
<td>0.00</td>
<td>0.89</td>
<td>0.00</td>
<td>1</td>
</tr>
<tr>
<td>0.00</td>
<td>0.89</td>
<td>0.00</td>
<td>1</td>
</tr>
</tbody>
</table>

It's clear that only the functions `get` and to a much lesser extent `put` are relevant to performance improvement.
Examine \{\text{get}\} and we find it traverses a linked list.

So replace it with a binary tree and the program runs 200x faster on War and Peace.

$ \text{clang} \ -O3 \text{ word\_frequency1.c} \ -o \text{ word\_frequency1}$
$ \text{time word\_frequency1} \ <\text{WarAndPeace.txt} \ >/\text{dev/null}$
real \quad 0m0.277s
user \quad 0m0.268s
sys \quad 0m0.008s

Was C the best choice for our count words program?
Shell, Perl and Python are slower - but a lot less code.

So faster to write, less bugs to find, easier to maintain/modify

```bash
$ time word_frequency1 <WarAndPeace.txt >/dev/null
real  0m0.277s
user  0m0.268s
sys   0m0.008s

$ time word_frequency.sh <WarAndPeace.txt >/dev/null
real  0m0.564s
user  0m0.584s
sys   0m0.036s

$ time word_frequency.pl <WarAndPeace.txt >/dev/null
real  0m0.643s
user  0m0.632s
sys   0m0.012s

$ time word_frequency.py <WarAndPeace.txt >/dev/null
real  0m1.046s
user  0m0.836s
```
Here is a cp implementation in C using low-level calls to read/write

```c
while (1) {
    char c[1];
    int bytes_read = read(in_fd, c, 1);
    if (bytes_read < 0) {
        perror("cp: ");
        exit(1);
    }
    if (bytes_read == 0)
        return;
    int bytes_written = write(out_fd, c, bytes_read);
    if (bytes_written <= 0) {
        perror("cp: ");
        exit(1);
    }
}
```
Here is a simple Perl program to calculate the n-th Fibonacci number:

```perl
sub fib {
    my ($n) = @_;  
    return 1 if $n < 3;  
    return fib($n-1) + fib($n-2); 
}
printf "fib(%d) = %d\n", _, fib(_)
```

It becomes slow near n=35.

```
$ time fib0.pl 35
fib(35) = 9227465
real   0m10.776s
user   0m10.729s
sys    0m0.016s
```

we can rewrite in C.
#include <stdio.h>

int fib(int n) {
    if (n < 3) return 1;
    return fib(n-1) + fib(n-2);
}

int main(int argc, char *argv[]) {
    for (int i = 1; i < argc; i++) {
        int n = atoi(argv[i]);
        printf("fib(%d) = %d\n", n, fib(n));
    }
}

Faster but the program’s complexity doesn’t change:

$ clang -O3 -o fib0 fib0.c
$ time fib0 45
fib(45) = 1134903170
real  0m4.994s
user  0m4.976s
Performance Improvement Example - Fibonacci

```perl
#!/usr/bin/perl -w

sub fib {
    my ($n) = @_; 
    return 1 if $n < 3;
    $f{$n} = fib($n-1) + fib($n-2) if !defined $f{$n}; 
    return $f{$n};
}

printf "fib(%d) = %d\n", $_, fib($_) foreach @ARGV;

It is very easy to cache already computed results in a Perl hash. 
This changes the program’s complexity from exponential to linear.

$time fib1.pl 45
fib(45) = 1134903170
real 0m0.004s
user 0m0.004s
sys 0m0.000s

Now for Fibonacci we could also easily change the program to an iterative, 
memoization is a general technique which can be employed in a variety of situations to improve performance.
```