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

Rapid development of a prototype good way to discover/assess performance issues.

Hence Fred Brooks maxim - “Plan To Throw One Away”
Where is execution time being spent?

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 -pg 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.

```bash
$ clang -O3 word_frequency0.c -o word_frequency0
$ time word_frequency0 <WarAndPeace.txt >/dev/null
real   0m52.726s
user   0m52.643s
sys    0m0.020s
```
clang (or gcc) can instrument the program to collect profiling information:

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

....

% cumulative self self total
time seconds seconds calls ms/call ms/call name
88.90 0.79 0.79 88335 0.01 0.01 get
7.88 0.86 0.07 7531 0.01 0.01 put
2.25 0.88 0.02 80805 0.00 0.00 get_word
1.13 0.89 0.01 1 10.02 823.90 read_words
0.00 0.89 0.00 2 0.00 0.00 size
0.00 0.89 0.00 1 0.00 0.00 create_map
0.00 0.89 0.00 1 0.00 0.00 keys
0.00 0.89 0.00 1 0.00 0.00 sort_words

....

Only the functions get and to a lesser extent put are relevant to performance improvement.
if we examine \texttt{get} we find it traverses a linked list.

\begin{verbatim}
void *get(map *m, char *key) {
    struct map_node *v;
    for (v = m->list; v != NULL; v = v->next) {
        if (strcmp(key, v->key) == 0) {
            return v->value;
        }
    }
    return NULL;
}
\end{verbatim}

this is an inefficient data structure (see COMP2521/COMP3121)

we can replace the linked list with a more efficient data structure, a binary tree
void *get(map *m, char *key) {
    return get_tree(m->tree, key);
}

static void *get_tree(struct map_tnode *t, char *key) {
    int compare;
    if (t == NULL)
        return NULL;
    compare = strcmp(key, t->key);
    if (compare == 0)
        return t->value;
    else if (compare < 0)
        return get_tree(t->smaller, key);
    else
        return get_tree(t->larger, key);
}
now the program runs 200x faster on War and Peace.

$ clang -O3 word_frequency1.c -o word_frequency1
$ time word_frequency1 <WarAndPeace.txt >/dev/null
real  0m0.277s
user  0m0.268s
sys   0m0.008s

Was C the best choice for our count words program?
while ($line = <>)
{
    $line =~ tr/A-Z/a-z/;
    foreach $word ($line =~ /[a-z]+/g) {
        $count{$word}++;
    }
}

@words = keys %count;
@sorted_words = sort { $count{$a} <=> $count{$b} } @words;
foreach $word (@sorted_words)
{
    printf "%8d %s\n", $count{$word}, $word;
}

source code for word_frequency.pl
```
tr -c a-zA-Z ' ' |
tr ' ' '\n'| 
tr A-Z a-z|
egrep -v '^$'| 
sort|
uniq -c
```

source code for word_frequency.sh
import fileinput, re, collections

count = collections.defaultdict(int)

for line in fileinput.input():
    for word in re.findall(r'\w+', line.lower()):
        count[word] += 1

words = count.keys()

sorted_words = sorted(words, key=lambda w: count[w])

for word in sorted_words:
    print("%8d %s" % (count[word], word))

source code for word_frequency.py
• Shell, Perl and Python are slower - but a lot less code.
• 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
sys    0m0.024s
$ wc word_frequency*.*
  286   759  3912  word_frequency1.c
     8    19     82  word_frequency.sh
    11    38    325  word_frequency.py
    14    43    301  word_frequency.pl
```
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", $_, fib($_) foreach @ARGV;
```

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 ....
Performance Improvement Example - Fibonacci - C

```c
#include <stdio.h>
#include <stdlib.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));
    }
    return 0;
}
```

source code for fib0.c

Faster but still slows quickly on larger inputs:

```
$ clang -O3 -o fib0 fib0.c
$ time fib0 45
fib(45) = 1134903170
real 0m4.994s
user 0m4.976s
sys 0m0.004s
```
sub fib {
  my ($n) = @_;
  return 1 if $n < 3;
  return $fib{$n} || ($fib{$n} = fib($n-1) + fib($n-2));
}

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

- very easy to cache already computed results in a Perl hash.
- changes execution time growth from exponential to linear.

$ time fib1.pl 45
fib(45) = 1134903170
real 0m0.004s
user 0m0.004s
sys 0m0.000s
• for Fibonacci we could also easily change the program to an iterative form which would be linear too.

• memoization is a general technique which can be employed in a variety of situations to improve performance.