Most Wondrous - Optimisation Techniques

Jonathan Lam
Tutor: Lucy Qiu

August 18, 2018

In hindsight, I probably would have saved more time by running my code from the beginning and letting it run in the background, rather than spend 18 hours finding small optimisation tricks. Regardless, I did learn a lot including how compilers optimise C code and other optimisation techniques.

1 Normal Code

Here is my 'normal code' - unoptimised and just plain brute force.

```
int main(void) {
  unsigned long long i = 1;
  while (count(i) < 1234) {
    ++i;
  }
  printf("%llu has length %d\n", i, count(i));
  return EXIT_SUCCESS;
}

int count(unsigned long long num) {
  unsigned long long i = num;
  int counter = 1;
  while (i != 1) {
    if (i % 2 == 0) {
      i = i / 2;
    } else {
      i = 3*i + 1;
    }
    counter++;
  }
  return counter;
}
```

It is pretty obvious what I am doing here. The function `count` takes a number as an input, and returns its length. The `main` function starts at 1 and calculates the count for each number and does this for every number until the count is greater than 1234. This plain approach would take approximately 63 hours to run. My optimisations have allowed me to improve efficiency and search through the numbers 5.25 times faster than a standard approach. My approach allowed me to save 51 hours of computation time.

By the way, the answer is $\boxed{133,561,134,663}$ with a length of 1235 (which is the first number greater than 1234).

2 Arithmetic Tricks

This is stuff like replacing powers with multiplication (this is cheaper for computers) or replacing multiplication with addition (much more cheaper). So I replaced `3*i + 1` with `i+i+i + 1`.

Bitwise operators
I also experimented with using bitwise operators since shifting bits is supposedly faster than multiplication. However, during experimentation (I ran the bash command `time ./mostWondrous`), there was no evidence to support that the bitwise tricks helped. It seems like the compiler did a good job at optimising this anyway.

However, I did learn a lot about bitwise operations in C. Notably **shifting** to double and divide by powers of 2.

e.g. I tried using `i = i >> 1` (which could be further shortened to `i >>= 1`) in place of `i = i / 2`.
I also tried `i = (i << 1) + i + 1` instead of `3*i + 1`.

Parity Checking
Instead of `if (i % 2 == 1)` (checks for odd numbers), I tried `if (n & 1)` (this evaluates to 1 if the number is odd, and uses the bitwise AND)

All of these improvements were rather superficial (saves a couple of seconds at best) so I had to look for better ideas.

3 Introducing some Terminology

We'll first define a few functions and terms, because I realise explaining my method without this to be too cumbersome.

Let the *trajectory* of a number be the sequence of numbers following the rule. For example, for the number 10, we have

$$10 \to 5 \to 16 \to 8 \to 4 \to 2 \to 1$$

and so its trajectory would be the sequence of these numbers.

Let the function count(n) be the number of steps it takes for n to reach 1. For example, count(10) = 7 and count(8) = 4.

Let a number n be called a *count record* if it is the smallest number such that count(n) > count(m) for all $m < n$ (i.e. such that it has a higher count than any numbers before it).

4 Eat up my RAM (arrays and memory)

My next thought was to store the *count* (i.e. length) of previously calculated values.

For example, the *trajectory* of 10 is given by

$$10 \to 5 \to 16 \to 8 \to 4 \to 2 \to 1$$

and has a *count* of 7. So now, next time we need to calculate the count of 10, we can just pull that value out of memory rather than having to calculate the remaining numbers. We can see why this is particularly useful when the numbers are in the billions and their lengths are several hundred.

For example, to calculate the *count* of 17, this is

$$17 \to 52 \to 26 \to 13 \to 40 \to 20 \to 10$$

and we can pretty much stop there because we know that once we arrive at 10, we just add a 7 on top of that. So $\text{count}(17) = 6 + 7 = 13$

I decided to build memory for values of i from 1 until 2,298,025 (length 560). I thought that this would be a decent trade off because later on, when the numbers are in the hundreds of billions, I would rather look up these values than calculate them again.

```
#define MEMORY_SIZE 1000000
unsigned long long memory[MEMORY_SIZE] = {0};

void buildMemory(void) {
  unsigned long long i = 1;
  while (count(i) < 560) {
    ++i;
  }
}
```

I also modified the `count` function.

```
int count(unsigned long long num) {
  unsigned long long i = num;
  int counter = 1;
  while (i != 1) {
    if (i % 2 == 0) {
      i = i / 2;
    } else { // everything above this line was the same as before
      i = i+i+i + 1; // addition trick
    }

  if (i<MEMORY_SIZE && memory[i] != 0) {
    counter += memory[i]; // Add stored count from memory onto current count
    i = 1; //break loop. We have finished
  } else {
    counter++;
  }

  if (num<MEMORY_SIZE) {
    memory[num] = counter; // Store new value in memory
  }
}
```

This has managed to improve my speed by a fair bit, although the next section will improve speed by like 80%.

5 Narrowing down the search cases

I will prove that we can skip certain numbers, as these are guaranteed to not be the answer. This will certainly speed up the search.

First of all, we can skip checking the even numbers. This improves the speed by 50% as we only need to check the odd numbers. The explanation for this is provided in section 5.1.

Theorem 1. If $a > b$ and $\text{count}(a) < \text{count}(b)$, then we can skip checking a.
Explanation. Since a is bigger than b, but has a smaller count, it cannot hold the record as we have found a lower number with a higher count.
Alternate wording: If $a > b$ and a lies on the trajectory of b, then we can skip a.

2 mod 3 Let's start with an odd number, $2x + 1$. Then if we apply the sequence, we get

$$2x + 1 \to 3(2x + 1) + 1 = 6x + 4 \to 3x + 2$$

hence we can skip checking numbers of the form $3x + 2$ because we know that a smaller number $(2x + 1)$ has a higher count. (theorem 1).

Numbers mod 9
We can skip numbers which are congruent to 2, 4, 5 or 8 mod 9.

- **2 mod 9** Consider $6x + 1 \to 18x + 4 \to 9x + 2$.
  So for example, if we start with 7, then we have trajectory $7 \to 22 \to 11 \to \ldots$. We can skip checking 11 because it lies on the trajectory of 7.
  $\text{count}(7) = \text{count}(11) + 2 \implies \text{count}(7) > \text{count}(11)$.

  Hence since $\text{count}(9x + 2) < \text{count}(6x + 1)$ and $9x + 2 > 6x + 1$, we cannot have $9x + 2$ being a count record.
- **4 mod 9** Consider $8x + 3 \to 24x + 10 \to 12x + 5 \to 36x + 16 \to 18x + 8 \to 9x + 4$
- **5 mod 9** Consider $6x + 3 \to 18x + 10 \to 9x + 5$
- **8 mod 9** Consider $6x + 5 \to 18x + 16 \to 9x + 8$

All these are due to theorem 1.

Theorem 2. If $a > b$ and $\text{count}(a) = \text{count}(b)$, then we can skip checking a.
Explanation. Since a is bigger than b, but has a smaller count, it cannot hold the record as we have found a lower number with the same count.

Using this, we can skip numbers which are equal to 5 mod 8.
$8x + 5 \to 24x + 16 \to 12x + 8 \to 6x + 4 \to \cdots$
$\quad 8x + 4 \to 4x + 2 \to 2x + 1 \to 6x + 4 \nearrow$

Hence $\text{count}(8x + 5) = 3 + \text{count}(6x + 4) = \text{count}(8x + 4)$.
The two trajectories merge, so we can ignore the larger of these numbers.

5.1 Even Numbers

Is it possible for an even number to be a *count record*? Yes, but we can actually work these out by hand.

First we have this 'theorem'. It comes from the sequence an even number forms.

$$2k \to k \to \cdots$$

Hence we have $\text{count}(2k) = \text{count}(k) + 1$ (theorem 3). We will use this property to prove this next important result.

If n is a count-record, then $2n$ is the least even number that can be a count-record. Hence we can skip checking all the even numbers in between n and $2n$. We just need to check (some of) the odd numbers in between and find their counts. The only time when $2n$ would be a count-record is if $\text{count}(n+m) \leq \text{count}(n)$ for all $0 < m < n$.

So then my tutor (Lucy) asks, 'is it possible for an even number in between n and $2n$ to have a greater count?' and I thought about how to prove it. Whilst standing on the bus, I thought of it!

Let $2j$ be an even number between n and $2n$. Hence $2j < 2n \implies j < n$. We have $\text{count}(j) < \text{count}(n)$ by definition (if n is a count-record, then the count for any number below it must be less).

$$\begin{aligned} \text{count}(2j) &= 1 + \text{count}(j) \\ &< 1 + \text{count}(n) \\ &= \text{count}(2n) \end{aligned}$$

To summarise:

- If n is a count-record, then we only even number we need to check is $2n$.
- $2n$ is the upper bound of the next count-record
- We only need to check the odd numbers in between n and $2n$.

6 Other considerations

If we have previously visited a number, then it is guaranteed to have a lower count, and we can skip checking that number. I intended on creating a boolean array that stores whether we have visited that number already. However, I need an array of size billions and not sure how to do that...

7 Binary Search

The idea of using a binary search was proposed by another student in my tutorial/lab. The idea is, instead of checking every single number one by one, we could check a number, skip a few (e.g. by doubling the number) and check that number. And we keep doing this until we find a number with a length greater than 1234. Then we have found an upper bound for the answer. We can then work backwards and only check the numbers in between our upper and lower bounds.

Unfortunately, this technique would only work if the function was monotonic. That is, if $a > b$, then count(a) $>$ count(b). This is clearly not true by looking at small cases.

What a shame...

There does not seem to be any other similar technique that allows us to skip values or narrow down the answer.