# COS125 - Precept 5 (Performance)

## 1 Tracing Loops

Write the largest value that variable `counter` takes on in each of the code snippets below and write the function of $n$ that corresponds to this value. You may assume that $n$ is a power of 2.

| Code | $n = 2$ | $n = 8$ | $n = 128$ | $f(n)$ |
|---|---|---|---|---|
| `int counter = 0;`<br>`for (int i = 0; i < n; i++)`<br>`   counter++;` | 2 | 8 | 128 | $n$ |
| `for (int i = 0, counter = 1; i < n; i++)`<br>`   counter *= 2;` | | | | |
| `int counter = 0;`<br>`while (counter < n)`<br>`   counter++;` | | | | |
| `int counter = 0;`<br>`for (int i = 1; i <= n; i *= 2)`<br>`   counter++;` | | | | |
| `for (int i = 0, counter = 0; i < n; i++)`<br>`   for (int j = 0; j < n; j++)`<br>`      counter++;` | | | | |
| `int counter = 0;`<br>`for (int i = 1; i <= n; i *= 2)`<br>`   for (int j = 0; j < n; j++)`<br>`      counter++;` | | | | |

## 2 Image Processing

Download `precept5.zip` from the precepts webpage; unzip and open the project folder. Open `Negative.java`, compile and run it on the 5 images in the folder with the `-Xint` option.[1] Write down the elapsed time for each of them in the table below.

| `1.jpg` | `2.jpg` | `3.jpg` | `4.jpg` | `5.jpg` |
|---|---|---|---|---|
| | | | | |

[1]Running `java-introcs -Xint Negative filename.png` disables optimizations by the Java Virtual Machine. This allows you to see the difference between a (non-optimized) inefficient implementation and a more efficient one.

Now, copy `Negative.java` into another file (choose any suitable name you like). Update this new file to use the functions `StdPicture.getARGB()` (which receives two arguments – integer column and row values – and returns an RGB color encoded into an `int`) and `StdPicture.setARGB()` (which receives three arguments: row and column values, as well as an RGB value encoded into an `int`) instead of `StdPicture.getRed/Green/Blue()` and `StdPicture.setRGB()`.

Use the expression `16777215 ^ rgb` to compute the negative of a color `rgb` encoded as an `int`.[2] Fill in the table below with the elapsed times of this alternative implementation (use the `-Xint` option again to get a fair comparison).

| 1.jpg | 2.jpg | 3.jpg | 4.jpg | 5.jpg |
|---|---|---|---|---|
| | | | | |

# 3 Primes & Factoring

## Less-Naive Factoring

We have seen a simple but inefficient algorithm for factoring in lecture: just try to divide the input $n$ by all numbers from 1 to $n$ (its potential divisors), recording successful divisions.

Compile `Factors.java` and run it on the following `long` command-line arguments. Record the time taken for each one of them.

| 797026819 | 1594053611 | 6376214287 | 12752428583 | 25504857179 |
|---|---|---|---|---|
| | | | | |

Now, copy `Factors.java` into another file (choose any suitable name you like). Implement the improvement, mentioned in the lecture slides, that only tries to divide by numbers up to $\sqrt{n}$ (and if none succeed, concludes that $n$ is the only nontrivial factor). Run it on the same sequence of inputs and record the new elapsed time for each of them.

| 797026819 | 1594053611 | 6376214287 | 12752428583 | 25504857179 |
|---|---|---|---|---|
| | | | | |

Finally, run both versions of `Factor` on `2541006914742139321`. Can you explain the difference in performance (or lack thereof)?

[2] `rgb` encodes the red channel in its 8 least-significant bits, green in bits 9 to 16, blue in 17 to 24, and alpha (which controls transparency) in bits 25 to 32. The operator `^` is the bitwise XOR, which enables subtracting 255 from all three channels (leaving alpha unchanged) in a single integer operation: the bitwise XOR with the number whose binary representation is a sequence of 24 ones, i.e., 16777215.

## Repeated Squaring

An important primitive for cryptography is modular exponentiation: given a positive integer $n$, a base $b < n$ and an exponent $e < n$, the goal is to compute $b^e \pmod{n}$ (the remainder of the division of $b^e$ by $n$).[3]

First, fill in the program `ModularExp.java` so that it takes 3 `long` command-line arguments, interprets them as $b$, $e$ and $n$, and computes $b^e \pmod{n}$. Notice that computing $b^e$ may cause a `long` overflow, so your code should take care to avoid it. (*Hint:* `a * (a % b) = (a * a) % b`.)

Write the time taken to compute $b^e \pmod{n}$ for the values in the table below.

| $b$ | $e$ | $n$ | **Time** (sec) |
|---|---|---|---|
| 35924 | 50000000 | 200830686 | |
| 28075 | 100000000 | 280308297 | |
| 605 | 200000000 | 898221318 | |
| 97658 | 400000000 | 586182711 | |
| 59377 | 800000000 | 599830768 | |
| 10798 | 1600000000 | 600400252 | |

Now, copy `ModularExp.java` into a new file (with any suitable name of your choice) and modify the program using the strategy of *repeated squaring*: instead of multiplying a variable by $b$ a total of $c$ times, multiplying the variable *by itself* $k$ times computes $b^{2^k}$; therefore, we only need $\lceil \log c \rceil$ iterations (rather than $c$ – an *exponential* improvement!).

More precisely, the repeated squaring algorithm to compute $b^e$ is as follows: initialize the variables `result` to 1, `power` to the base $b$ and `e` to the exponent $e$. Then repeat the following as long as `e` $> 0$:

1. If $e$ is odd, set `result` to `result * power`.
2. Set `e` to `e`/2, rounding down.
3. Set `power` to `power * power`.

At the end of this loop, the variable `result` is equal to $b^e$.[4]

Now fill in the table below with the runtimes of your new algorithm.

## Fermat's "Primality" Test (Bonus)

You can now put repeated squaring strategy to good use: testing if a number is prime! (Sort of.)

The test is inspired by the following identity, known as Fermat's Little Theorem:[5] for any prime number $p$ and $a < p$, we have $a^p \equiv a \pmod{p}$.

[3]While modular arithmetic might look strange at first, we're all quite used to it: we know that 5 hours after 11am is 4pm because $11 + 5 \equiv 4 \pmod{12}$.

[4]This is because `result` is multiplied by $b^{2^k}$ if and only if $2^k$ appears in the binary representation $\sum_k 2^k$ of $e$, so `result` $= \prod_k b^{2^k} = b^{\sum_k 2^k} = b^e$.

[5]Not to be confused with Fermat's *Last* "Theorem," which took 400 years to finally prove – in Princeton!

| $b$ | $e$ | $n$ | **Time** (sec) |
|---|---|---|---|
| 35924 | 50000000 | 200830686 | |
| 28075 | 100000000 | 280308297 | |
| 605 | 200000000 | 898221318 | |
| 97658 | 400000000 | 586182711 | |
| 59377 | 800000000 | 599830768 | |
| 10798 | 1600000000 | 600400252 | |

This leads to the following strategy to check if a number $n$ is prime: sample a random number $a < n$ and compute $a^n \pmod{n}$: if the result is not $a$, then we know $n$ cannot be prime. (However, $a^n \equiv \pmod{n}$ does not guarantee $n$ is prime, as we'll see next.)

Write a program `FermatTest.java` that takes two `long` command-line arguments $n$ and $k$, and runs $k$ iterations of Fermat's test on the number $n$. Remember that even if a single test fails, $n$ cannot be prime; and if all tests pass, $n$ is either "special" or prime. (You may find the function `StdRandom.uniformLong()` useful.)

Run `FermatTest` on the pairs $(n, k)$ below and report whether $n$ passes (every) Fermat test. Then run `Factor` on those that passed to figure out if they're really prime or not.

| $n$ | $k$ | **Passed?** |
|---|---|---|
| 10000 | 10000 | |
| 986088961 | 10000 | |
| 986088977 | 10000 | |
| 1177800343 | 10000 | |
| 1177800481 | 10000 | |