Writeup 1

The performance bottleneck that I found in isort implementation was the line `while (index >= left && *index > val)` in the isort function. Perf noted that the instruction `cmp %edx,%esi` took more than 90% of the time to run, which corresponds to the comparison between `val` and the value at `index`, which is an indeterministic comparison.

Writeup 2

Considering the fact that the ./sum program is randomly accessing all locations in a 40MB large array, I'm surprised to find that the D1 miss rate is only 16.7% while there's very little LLd and LL miss rate.

```
$ srun -w agate-1 -t 5 --reservation=eec289q valgrind --tool=cachegrind --branch-sim=yes ./sum
......
Allocated array of size 10000000
Summing 100000000 random values...
Done. Value = 938895920
......
==1001381== D1  miss rate:          16.7% (       25.0%     +         0.3%  )
==1001381== LLd miss rate:           0.1% (        0.0%     +         0.3%  )
==1001381==
......
==1001381== LL miss rate:            0.0% (        0.0%     +         0.3%  )
......
```

However, when I tried to increase the size of the array to 10x and 30x the original size, there's very little increase in D1 miss rate but significant increaes in LLd and LL miss rate.

```
$ srun -w agate-1 -t 5 --reservation=eec289q valgrind --tool=cachegrind --branch-sim=yes ./sum
......
Allocated array of size 300000000
Summing 100000000 random values...
Done. Value = -1971034320
......
==1004301== D1  miss rate:          18.6% (       25.0%     +         7.9%  )
==1004301== LLd miss rate:          15.4% (       19.8%     +         7.9%  )
==1004301==
......
==1004301== LL miss rate:            2.2% (        1.9%     +         7.9%  )
......
```

Project Matrix Rotation

The approach I chose to optimize the rotate.c implementation focuses on addressing two of the performance bottlenecks: read and write as many bits in a single instruction as possible, and reduce cache miss rate from load instructions. Therefore, I scaled up the original 4-quadrant in-place rotation algorithm from moving 1 bit at a time to moving and rotating a block of 32x32 bits at a time. This allows more parts of the code to assign 32 bits in each instruction, and it also means cross-column access on the matrix is reduced by 32 times. I've also tried implementing rotate using block size of 64, but I didn't observe any performance inprovements, and the fact that the matrix's quadrants aren't always evenly divisible into 64x64 blocks would result in more difficult edge case handling. This is why I chose 32x32 block size in the end.

The program can consistently reach up to tier 58 (although it has reached tier 60 during a particular time frame, which is likely due to variations in the server load). From the comparison between the runtime of the stock program and my implementation for matrix size equivalent to tier 37, the speedup of my implementation compared to the stock program is 13.21 times.

As for the correctness test set and some larger randomly generated matrix rotation cases, I haven't noticed any correctness issue with my most recent implementation.

From the perf analysis output, I believe my performance bottleneck is from the cache misses when accessing matrix content in different rows, since the report shows that these 2 vmovd instructions, which are involved in reading the matrix content into the registers, take up more than 60% of the execution time.

```
31.96 │       vmovd       %r15d,%xmm8
...
32.22 │       vmovd       %r10d,%xmm8
...
```