Homework & Lab:

Week2:

Buildwords:

```
!/bin/bash
export LC_ALL='C'
grep '<td>' \
| tr '[:upper:]' '[:lower:]'\
| sed 's/`/'\''/g'\
| sed 's/<[^>]*>//g'\
| tr -s '[:space:]'\
| sed -n 2~2p\
| sed  's/,/\n/g'\
| sed  's/ /\n/g'\
| sed 's/\r//g'\
| tr -s '[:space:]' \
| sed '/[^a^e^i^o^p^k^m^n^l^w^h^u^'\'']/d'\
| sort -u
```

Sameln:

```
#!/bin/bash
export LC_ALL='C'
dir=$1
cd "$dir"
prior=`ls .[^.]* | sort -f`
remain=`ls | sort -f`
declare -a ARRAY
count=0

for file in $prior
do
  if [ ! -L "$file" ]&&[ -f "$file" ]&&[ -r "$file" ]
  then
       ARRAY[$count]="$file"
       let count=count+1
  fi
done

for file in $remain
do
  if [ ! -L "$file" ]&&[ -f "$file" ]&& [ -r "$file" ]
  then
```

```
      ARRAY[$count]="$file"
      let count=count+1
  fi
done

index=0
while [ $index -lt $[count-1] ]
do
   let next=index+1
   while [ $next -lt $count ]
   do
      cmp -s "${ARRAY[$index]}" "${ARRAY[$next]}"
      if [ $? -eq 0 ]
      then
         ln -f "${ARRAY[$index]}" "${ARRAY[$next]}"
         break
      fi
      let next=next+1
   done
   let index=index+1
done
```

Week3:

Comm.py:

```
#!/usr/local/cs/bin/python3

import random, sys
from optparse import OptionParser

class randline:
  def __init__(self, filename):
    self.lines = []
    if filename == '-':
       self.lines.append(sys.stdin.readline())
    else:
       f = open (filename, 'r')
       self.lines = f.readlines()
       f.close()

def main():
    version_msg = "%prog 2.0"
    usage_msg = """%prog [OPTION]...
compare two files."""
```

```
    parser = OptionParser(version=version_msg,  usage=usage_msg)
    parser.add_option("-1", "--suppressfile1",
action="store_true",\
 dest="choice1",default=False, help="suppress lines unique to
file1")
    parser.add_option("-2", "--suppressfile2",
action="store_true",\
 dest="choice2", default=False, help="suppress lines unique to
file2")
    parser.add_option("-3", "--suppresscommon",
action="store_true",\
 dest="choice3", default=False, help="suppress duplicated lines
in both files")
    parser.add_option("-u", "--unsorted", action="store_true",\
 dest="choice4", default=False, help="compare unsorted files")

    options, args = parser.parse_args(sys.argv[1:])

    try:
        choice1 = bool(options.choice1)
    except:
        parser.error("invalid CHOICE:
{False}".format(options.choice1))
    try:
        choice2 = bool(options.choice2)
    except:
        parser.error("invalid CHOICE:
{False}".format(options.choice2))
    try:
        choice3 = bool(options.choice3)
    except:
        parser.error("invalid CHOICE:
{False}".format(options.choice3))
    try:
        choice4 = bool(options.choice4)
    except:
        parser.error("invalid CHOICE:
{False}".format(options.choice4))
    if len(args) != 2:
        parser.error("wrong number of operands")
        sys.exit
    if args[0] == "-" and args[1] == "-":
        parser.error("There should not be two files from stdin")
```

```
        sys.exit
    if args[0] == args[1]:
        parser.error("Two same operands are not permitted")
        sys.exit

    handle1 = randline(args[0])
    handle2 = randline(args[1])
    if handle1.lines[-1][-1] != "\n":
      handle1.lines[-1] = handle1.lines[-1] + "\n"
    if handle2.lines[-1][-1] != "\n":
      handle2.lines[-1] = handle2.lines[-1] + "\n"
    handleunsorted1 = handle1.lines
    handleunsorted2 = handle2.lines
    handletemp1 = handle1.lines
    handletemp2 = handle2.lines
    if choice4 == False:
       if sorted(handle1.lines) != handle1.lines or
sorted(handle2.lines)\
 != handle2.lines:
          parser.error("Files cannot be unsorted")
          sys.exit
       else:
          total = []
          for lines in handle1.lines:
             total.append(lines)
          for lines in handle2.lines:
             total.append(lines)
          total.sort()

          if choice1 == True and choice2 == True and choice3 == False:
            for lines in total:
              if lines in handletemp1 and lines in handletemp2:
                sys.stdout.write (lines)
                handletemp1.remove(lines)
                handletemp2.remove(lines)
              elif lines in handletemp1 and lines not in handletemp2:
                handletemp1.remove(lines)
              elif lines in handletemp2 and lines not in handletemp1:
                handletemp2.remove(lines)

          if choice1 == True and choice2 == False and choice3 == True:
            for lines in total:
              if lines in handletemp1 and lines in handletemp2:
                handletemp1.remove(lines)
```

```
            handletemp2.remove(lines)
          elif lines in handletemp1 and lines not in handletemp2:
            handletemp1.remove(lines)
          elif lines in handletemp2 and lines not in handletemp1:
            sys.stdout.write (lines)
            handletemp2.remove(lines)

        if choice1 == False and choice2 == True and choice3 == True:
          for lines in total:
            if lines in handletemp1 and lines in handletemp2:
              handletemp1.remove(lines)
              handletemp2.remove(lines)
            elif lines in handletemp1 and lines not in handletemp2:
              sys.stdout.write(lines)
              handletemp1.remove(lines)
            elif lines in handletemp2 and lines not in handletemp1:
              handletemp2.remove(lines)

        if choice1 == True and choice2 == False and choice3 ==
False:
          for lines in total:
            if lines in handletemp1 and lines in handletemp2:
              sys.stdout.write ("   \t" + lines)
              handletemp1.remove(lines)
              handletemp2.remove(lines)
            elif lines in handletemp1 and lines not in handletemp2:
              handletemp1.remove(lines)
            elif lines in handletemp2 and lines not in handletemp1:
              sys.stdout.write (lines)
              handletemp2.remove(lines)

        if choice1 == False and choice2 == True and choice3 ==
False:
          for lines in total:
            if lines in handletemp1 and lines in handletemp2:
              sys.stdout.write ("   \t" + lines)
              handletemp1.remove(lines)
              handletemp2.remove(lines)
            elif lines in handletemp1 and lines not in handletemp2:
              sys.stdout.write(lines)
              handletemp1.remove(lines)
            elif lines in handletemp2 and lines not in handletemp1:
              handletemp2.remove(lines)
```

```
        if choice1 == False and choice2 == False and choice3 ==
True:
          for lines in total:
            if lines in handletemp1 and lines in handletemp2:
              handletemp1.remove(lines)
              handletemp2.remove(lines)
            elif lines in handletemp1 and lines not in handletemp2:
              sys.stdout.write(lines)
              handletemp1.remove(lines)
            elif lines in handletemp2 and lines not in handletemp1:
              sys.stdout.write ("    \t" + lines)
              handletemp2.remove(lines)

        if choice1 == False and choice2 == False and choice3 ==
False:
          for lines in total:
            if lines in handletemp1 and lines in handletemp2:
              sys.stdout.write ("    \t    \t" + lines)
              handletemp1.remove(lines)
              handletemp2.remove(lines)
            elif lines in handletemp1 and lines not in handletemp2:
              sys.stdout.write(lines)
              handletemp1.remove(lines)
            elif lines in handletemp2 and lines not in handletemp1:
              sys.stdout.write ("    \t" + lines)
              handletemp2.remove(lines)


    elif choice4 == True:
        if choice1 == True and choice2 == True and choice3 == False:
          for lines in handleunsorted1:
            if lines in handletemp2:
              sys.stdout.write (lines)
              handletemp2.remove(lines)

        if choice1 == True and choice2 == False and choice3 == True:
          for lines in handleunsorted1:
            if lines in handletemp2:
              handletemp2.remove(lines)
          for lines in handletemp2:
              sys.stdout.write (lines)

        if choice1 == False and choice2 == True and choice3 == True:
          for lines in handleunsorted1:
```

```
            if lines in handletemp2:
              handletemp2.remove(lines)
            else:
              sys.stdout.write(lines)

        if choice1 == True and choice2 == False and choice3 ==
False:
          for lines in handleunsorted1:
            if lines in handletemp2:
              sys.stdout.write ("    \t" + lines)
              handletemp2.remove(lines)
          for lines in handletemp2:
              sys.stdout.write (lines)

        if choice1 == False and choice2 == True and choice3 ==
False:
          for lines in handleunsorted1:
            if lines in handletemp2:
              sys.stdout.write ("    \t" + lines)
              handletemp2.remove(lines)
            else:
              sys.stdout.write(lines)

        if choice1 == False and choice2 == False and choice3 ==
True:
          for lines in handleunsorted1:
            if lines in handletemp2:
              handletemp2.remove(lines)
            else:
              sys.stdout.write(lines)
          for lines in handletemp2:
              sys.stdout.write ("    \t" + lines)

        if choice1 == False and choice2 == False and choice3 ==
False:
          for lines in handleunsorted1:
            if lines in handletemp2:
              sys.stdout.write ("    \t    \t" + lines)
              handletemp2.remove(lines)
            else:
              sys.stdout.write(lines)
          for lines in handletemp2:
              sys.stdout.write ("    \t" + lines)
```

```
if __name__ == "__main__":
   main()
```

Week5:

Srot13.c:

```
#include <stdio.h>
#include <stdlib.h>

int rot13cmp(const char* a, const char* b)
{

  for(;;)
  {
      int aval = (int)(*a);
      int bval = (int)(*b);

      if ((aval >= 65 && aval <= 77) || (aval >= 97 && aval <= 109))
          aval += 13;
      else if ((aval >= 78 && aval <= 90) || (aval >= 110 && aval
<= 122))
          aval -= 13;

      if ((bval >= 65 && bval <= 77) || (bval >= 97 && bval <= 109))
         bval += 13;
      else if ((bval >= 78 && bval <= 90) || (bval >= 110 && bval
<= 122))
         bval -= 13;

      if (aval == '\n' && bval == '\n')
         return 0;
      else if (aval == '\n')
         return -1;
      else if (bval == '\n')
         return 1;


      if (aval == bval)
      {
         a++;
         b++;
      }
      else
      {
```

```
        return (aval - bval);
      }
    }
}

int rot13cmpptr(const void* s1, const void* s2)
{
  return rot13cmp(*(const char**)s1, *(const char**)s2);
}

int main()
{
    int size = 2048;
    char* c = (char*)malloc(sizeof(char) * size);
    if (c == NULL)
    {
        fprintf(stderr, "Unable to allocate memory");
        exit(1);
    }

    int thischar;
    int i = 0;
    int numOfLines = 0;

    for (;;)
    {
        thischar = getchar();
        if (thischar == EOF)
        {
            break;
        }

        c[i] = thischar;
        i++;
        if (i == size)
        {
            size += 2048;
            c = (char*)realloc(c, size);
            if (c == NULL)
            {
                fprintf(stderr, "Unable to allocate memory");
                exit(1);
            }
        }
```

```
}
if (i == 0)
{
    fprintf(stderr, "Blank file");
    exit(1);
}

int jk = 0;
for(;jk<i; jk++)
{
    if (c[jk]== '\n')
    {
        numOfLines++;
    }
}


if (*(c + i - 1) != '\n')
{
    *(c + i) = '\n';
    i++;
    numOfLines++;
}

char** space = (char**)malloc(sizeof(char*) * numOfLines);
if (space == NULL)
{
    fprintf(stderr, "Unable to allocate memory");
    exit(1);
}

int j = 0;
char* cnew = c;
while (j < numOfLines)
{

    space[j] = cnew;
    for(;*cnew != '\n';cnew++)
    {}
    cnew++;
    j++;
}
qsort(space, numOfLines, sizeof(char*), rot13cmpptr);
int k = 0;
```

```
    for (; k < numOfLines; k++)
    {
        int m = 0;
        while (space[k][m] != '\n')
        {
            putchar(space[k][m]);
            m++;
        }
        putchar(space[k][m]);
    }
    free(c);
    free(space);
    return 0;
}
```

Week7:

Pthread:

```
#include "raymath.h"
#include "shaders.h"

#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <math.h>
#include <pthread.h>

struct imagepixel
{
  scene_t scene;
  int nthread;
  int thread_id;
};

static double dirs[6][3] =
{ {1,0,0}, {-1,0,0}, {0,1,0}, {0,-1,0}, {0,0,1}, {0,0,-1} };
static const int opposites[] = { 1, 0, 3, 2, 5, 4 };

static void
add_sphereflake( scene_t* scene, int sphere_id, int parent_id, int
dir,
        double ratio, int recursion_level )
{
    sphere_t* parent = &scene->spheres[parent_id];
```

```
    sphere_t* child = &scene->spheres[sphere_id];

    /* start at parents origin */
    mul( child->org, dirs[dir], (1.+ratio)*parent->rad );
    add( child->org, child->org, parent->org );
    child->rad = parent->rad * ratio;
    copy( child->color, parent->color );
    child->shader = parent->shader;
    scene->sphere_count++;
}

static int
recursive_add_sphereflake( scene_t* scene, int parent_id, int
parent_dir,
            int sphere_id, int dir,
            int recursion_level, int recursion_limit )
{
    const double ratio = 0.35;

    add_sphereflake( scene, sphere_id, parent_id, dir, ratio,
recursion_level );
    if( recursion_level > recursion_limit )
    {
        return sphere_id + 1;
    }

    /* six children, one at each cardinal point */
    parent_id = sphere_id;
    sphere_id = sphere_id + 1;
    for( int child_dir=0; child_dir<6; ++child_dir )
    {
        /* skip making spheres inside parent */
        if( parent_dir == opposites[child_dir] ) continue;
        sphere_id = recursive_add_sphereflake( scene, parent_id,
parent_dir,
                                               sphere_id, child_dir,
                                               recursion_level + 1,
                                               recursion_limit );
    }
    return sphere_id;
}

static scene_t
create_sphereflake_scene( int recursion_limit )
```

```
{
    scene_t scene;
    Vec3 color;
    sphere_t* sphere;

    init_scene( &scene );
    add_light( &scene, 2, 5, 0, 0.92, 0.76, 0.771 );
    add_light( &scene, -5, 3, -5, 0.96, 0.93, 0.88 );
    int max_sphere_count = 2 + powl( 6, recursion_limit + 2 );
    scene.spheres = realloc( scene.spheres,
                         max_sphere_count*sizeof( sphere_t ) );
    if( !scene.spheres )
    {
        fprintf( stderr, "Failed to get memory for sphereflake.
aborting.\n" );
        exit( -1 );
    }
    /* center sphere is special, child inherent shader and color
*/
    sphere = &(scene.spheres[0]);
    scene.sphere_count++;
    set( sphere->org, 0, -1, 0 );
    sphere->rad = 0.75;
    set( color, 0.75, 0.75, 0.75 );
    copy( sphere->color, color );
    sphere->shader = mirror_shader;
    recursive_add_sphereflake( &scene,
                           0, /* parent is the first sphere */
                           -1, /* -1 means no dir, make all children
*/
                           1, /* next free sphere index */
                           2, /* starting dir */
                           0, /* starting recursion level */
                           recursion_limit );

    return scene;
}

static void
free_scene( scene_t* arg )
{
    free( arg->lights );
    arg->light_count = 0;
    free( arg->spheres );
```

```
    arg->sphere_count = 0;
}

/******
 * Constants that have a large effect on performance */

/* how many levels to generate spheres */
enum { sphereflake_recursion = 3 };

/* output image size */
enum { height = 131 };
enum { width = 131 };

/* antialiasing samples, more is higher quality, 0 for no AA */
enum { halfSamples = 4 };
/******/

/* color depth to output for ppm */
enum { max_color = 255 };

/* z value for ray */
enum { z = 1 };

/* store the scaled_color parameter for the entire image */
float scaled_color_entire[width][height][3];

void multiThread(void* arg)
{
    struct imagepixel* ip = (struct imagepixel*) arg;

    Vec3 camera_pos;
    set( camera_pos, 0., 0., -4. );
    Vec3 camera_dir;
    set( camera_dir, 0., 0., 1. );
    const double camera_fov = 75.0 * (PI/180.0);
    Vec3 bg_color;
    set( bg_color, 0.8, 0.8, 1 );

    const double pixel_dx = tan( 0.5*camera_fov ) /
((double)width*0.5);
    const double pixel_dy = tan( 0.5*camera_fov ) /
((double)height*0.5);
    const double subsample_dx
    = halfSamples  ? pixel_dx / ((double)halfSamples*2.0)
```

```
    : pixel_dx;
    const double subsample_dy
    = halfSamples ? pixel_dy / ((double)halfSamples*2.0)
    : pixel_dy;

    /* for every pixel */
    for( int px=0; px<width; ++px )
    {
        const double x = pixel_dx * ((double)( px-(width/2) ));
        for( int py=ip->thread_id; py<height; py+=ip->nthread )
        {
            const double y = pixel_dy * ((double)( py-(height/2) ));
            Vec3 pixel_color;
            set( pixel_color, 0, 0, 0 );

            for( int xs=-halfSamples; xs<=halfSamples; ++xs )
            {
                for( int ys=-halfSamples; ys<=halfSamples; ++ys )
                {
                    double subx = x + ((double)xs)*subsample_dx;
                    double suby = y + ((double)ys)*subsample_dy;

                    /* construct the ray coming out of the camera,
through
                     * the screen at (subx,suby)
                     */
                    ray_t pixel_ray;
                    copy( pixel_ray.org, camera_pos );
                    Vec3 pixel_target;
                    set( pixel_target, subx, suby, z );
                    sub( pixel_ray.dir, pixel_target, camera_pos );
                    norm( pixel_ray.dir, pixel_ray.dir );

                    Vec3 sample_color;
                    copy( sample_color, bg_color );
                    /* trace the ray from the camera that
                     * passes through this pixel */
                    trace( &(ip->scene), sample_color, &pixel_ray,
0 );
                    /* sum color for subpixel AA */
                    add( pixel_color, pixel_color, sample_color );
                }
            }
```

```
          /* at this point, have accumulated (2*halfSamples)^2
samples,
           * so need to average out the final pixel color
           */
          if( halfSamples )
          {
             mul( pixel_color, pixel_color,
                (1.0/( 4.0 * halfSamples * halfSamples ) ) );
          }

          /* done, final floating point color values are in
pixel_color */
          float scaled_color[3];
          scaled_color[0] = gamma( pixel_color[0] ) * max_color;
          scaled_color[1] = gamma( pixel_color[1] ) * max_color;
          scaled_color[2] = gamma( pixel_color[2] ) * max_color;

          /* enforce caps, replace with real gamma */
          for( int i=0; i<3; i++)
             scaled_color[i] = max( min(scaled_color[i], 255),
0);

          /* write this pixel out to disk. ppm is forgiving about
whitespace,
           * but has a maximum of 70 chars/line, so use one line
per pixel
           */
          scaled_color_entire[px][py][0] = scaled_color[0];
          scaled_color_entire[px][py][1] = scaled_color[1];
          scaled_color_entire[px][py][2] = scaled_color[2];
       }
    }
}

int
main( int argc, char **argv )
{
    int nthreads = argc == 2 ? atoi( argv[1] ) : 0;

    if( nthreads < 1 )
    {
      fprintf( stderr, "%s: usage: %s NTHREADS\n", argv[0],
argv[0] );
      return 1;
```

```
    }

    scene_t scene =
create_sphereflake_scene( sphereflake_recursion );

    /* Write the image format header */
    /* P3 is an ASCII-formatted, color, PPM file */
    printf( "P3\n%d %d\n%d\n", width, height, max_color );
    printf( "# Rendering scene with %d spheres and %d lights\n",
           scene.sphere_count,
           scene.light_count );

    pthread_t* thread_mem = malloc(sizeof(pthread_t)*nthreads);
    struct imagepixel* ip = malloc(sizeof(struct
imagepixel)*nthreads);
    int ret;
    for ( int i=0; i<nthreads; i++ )
    {
        ip[i].thread_id = i;
        ip[i].scene = scene;
        ip[i].nthread = nthreads;
        ret = pthread_create(&thread_mem[i], NULL,
(void*)multiThread, &ip[i]);
        if (ret)
        {
       fprintf(stderr,"Can not create thread(s)\n");
         exit(-1);
        }
    }

    for ( int i=0; i<nthreads; i++ )
    {
        ret = pthread_join(thread_mem[i], NULL);
        if (ret)
        {
            fprintf(stderr,"Can not join thread(s)\n");
            exit(-1);
        }
    }

    for ( int px=0; px<width; ++px )
    {
        for ( int py=0; py<height; ++py )
        {
```

```
          printf( "%.0f %.0f %.0f\n",
                scaled_color_entire[px][py][0],
scaled_color_entire[px][py][1],
scaled_color_entire[px][py][2] );
       }
    }
    printf("\n");

    free_scene( &scene );
    free(thread_mem);
    free(ip);

    if( ferror( stdout ) || fclose( stdout ) != 0 )
    {
       fprintf( stderr, "Output error\n" );
     return 1;
    }

    return 0;
}
```

Week8:

Tr2b.c:

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, const char * argv[])
{
  /* Initialization */
  const char* from = argv[1];
  const char* to = argv[2];
  char tr[256];
  int i = 0;
  for (; i < 256; i++)
    {
      tr[i] = i;
    }

  /* Check duplicates in "from" */
  i = 0;
  for (; *(from + i); i++)
    {
      int j = i + 1;
```

```
      for (; *(from + j); j++)
        {
       if (from[i] == from[j])
           {
          fprintf(stderr, "'from' has duplicate bytes.\n");
          exit(1);
           }
        }
    }

  /* Check if sizes of "from" and "to" match */
  i = 0;
  while( *(from + i))
    {
      i++;
    }
  int k = 0;
  while( *(to + k))
    {
      k++;
    }
  if(i != k)
    {
      fprintf(stderr, "Operands have different sizes.\n");
      exit(1);
    }

  /* Transliteration */
  char c;
  i = 0;
  for (; *(from + i); i++)
    {
      tr[(int)from[i]] = to[i];
    }
  while ((c = getchar()) != EOF)
    {
      putchar(tr[c]);
    }
  return 0;
}
```

Tr2u.c:

```
#include <unistd.h>
#include <stdio.h>
```

```c
#include <stdlib.h>

int main(int argc, const char * argv[])
{
  /* Initialization */
  const char* from = argv[1];
  const char* to = argv[2];
  char tr[256];
  int i = 0;
  for (; i < 256; i++)
    {
      tr[i] = i;
    }

  /* Check duplicates in "from" */
  i = 0;
  for (; *(from + i); i++)
    {
      int j = i + 1;
      for (; *(from + j); j++)
        {
       if (from[i] == from[j])
           {
          fprintf(stderr, "'from' has duplicate bytes.\n");
          exit(1);
           }
        }
    }

  /* Check if sizes of "from" and "to" match */
  i = 0;
  while( *(from + i))
    {
      i++;
    }
  int k = 0;
  while( *(to + k))
    {
      k++;
    }
  if(i != k)
    {
      fprintf(stderr, "Operands have different sizes.\n");
      exit(1);
```

```
    }
  /* Transliteration */
  char in;
  char out;
  i = 0;
  for (; *(from + i); i++)
    {
      tr[(int)from[i]] = to[i];
    }
  while (read(0, &in, 1) != 0)
    {
      out = tr[in];
      write(1, &out, 1);
    }
  return 0;
}
```

Srot13u.c:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/stat.h>
#include <sys/types.h>

int cmp_count = 0;
int rot13cmp(const char* a, const char* b)
{
  cmp_count++;
  for(;;)
    {
      int aval = (int)(*a);
      int bval = (int)(*b);
      if ((aval >= 65 && aval <= 77) || (aval >= 97 && aval <= 109))
        aval += 13;
      else if ((aval >= 78 && aval <= 90) || (aval >= 110 && aval
<= 122))
        aval -= 13;

      if ((bval >= 65 && bval <= 77) || (bval >= 97 && bval <= 109))
        bval += 13;
      else if ((bval >= 78 && bval <= 90) || (bval >= 110 && bval
<= 122))
        bval -= 13;
```

```
     if (aval == '\n' && bval == '\n')
    return 0;
     else if (aval == '\n')
    return -1;
     else if (bval == '\n')
    return 1;

     if (aval == bval)
    {
        a++;
        b++;
    }
     else
    {
        return (aval - bval);
    }
   }
}

int rot13cmpptr(const void* s1, const void* s2)
{
  return rot13cmp(*(const char**)s1, *(const char**)s2);
}

int main(int argc, char** argv)
{
  struct stat st;
  fstat(0, &st);
  int buffer_size = 512;
  char* c;

  int thischar;
  int blank = 1;
  int i = 0;
  int numOfLines = 0;

  if (S_ISREG(st.st_mode))
    {
      int buffer_cap = st.st_size;
      if (buffer_cap == 0)
        {
       fprintf(stderr, "Number of comparisons: %d\n", 0);
       return 0;
        }
```

```
    else
      {
     c = (char*)malloc(sizeof(char) * buffer_cap);
     if (c == NULL)
         {
        fprintf(stderr, "Unable to allocate memory");
        exit(1);
         }
     for (;;)
         {
        fstat(0, &st);
        if (st.st_size != buffer_cap)
            {
       buffer_cap = st.st_size;
       c = (char*)realloc(c, sizeof(char) * buffer_cap);
       if (c == NULL)
                {
           fprintf(stderr, "Unable to allocate memory");
           exit(1);
                }
            }
        if (read(0, &thischar, 1) <= 0)
            {
       break;
            }
        if (thischar == '\n')
            {
       numOfLines++;
            }
        *(c + i) = thischar;
        i++;
        blank = 0;
         }
      }
  }
else
  {
    c = (char*)malloc(sizeof(char) * buffer_size);
    if (c == NULL)
      {
     fprintf(stderr, "Unable to allocate memory");
     exit(1);
      }
    for (;;)
```

```
     {
    if (read(0, &thischar, 1) <= 0)
         {
        break;
         }
    if (thischar == '\n')
         {
        numOfLines++;
         }
    *(c + i) = thischar;
    i++;
    blank = 0;
    if (i == buffer_size)
         {
        buffer_size *= 2;
        c = (char*)realloc(c, sizeof(char) * buffer_size);
        if (c == NULL)
             {
          fprintf(stderr, "Unable to allocate memory");
          exit(1);
             }
         }
     }
  }

if (blank)
  {
    fprintf(stderr, "Blank file");
    exit(1);
  }

if (*(c + i - 1) != '\n')
  {
    *(c + i) = '\n';
    i++;
    numOfLines++;
  }

char** space = (char**)malloc(sizeof(char*) * numOfLines);
if (space == NULL)
  {
    fprintf(stderr, "Unable to allocate memory");
    exit(1);
  }
```

```
  int j = 0;
  char* cnew = c;
  while (j < numOfLines)
    {

      space[j] = cnew;
      while (*cnew != '\n')
     {
          cnew++;
     }
      cnew++;
      j++;
    }
  qsort(space, numOfLines, sizeof(char*), rot13cmpptr);
  int k = 0;
  for (; k < numOfLines; k++)
    {
      int m = 0;
      while (space[k][m] != '\n')
        {
       write(1, &space[k][m], 1);
       m++;
        }
      write(1, &space[k][m], 1);
    }
  fprintf(stderr, "Number of comparisions: %d\n", cmp_count);
  free(c);
  free(space);
  return 0;
}
```

Week9:

Randmain.c:

```
#include <dlfcn.h>
#include <cpuid.h>
#include <errno.h>
#include <immintrin.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <randcpuid.h>
```

```
/* Main program, which outputs N bytes of random data.  */
int
main (int argc, char **argv)
{
  /* Check arguments.  */
  bool valid = false;
  long long nbytes;
  if (argc == 2)
    {
      char *endptr;
      errno = 0;
      nbytes = strtoll (argv[1], &endptr, 10);
      if (errno)
     perror (argv[1]);
      else
     valid = !*endptr && 0 <= nbytes;
    }
  if (!valid)
    {
      fprintf (stderr, "%s: usage: %s NBYTES\n", argv[0], argv[0]);
      return 1;
    }

  /* If there's no work to do, don't worry about which library to
use.  */
  if (nbytes == 0)
    return 0;

  /* Now that we know we have work to do, arrange to use the
     appropriate library.  */
  void (*initialize) (void);
  unsigned long long (*rand64) (void);
  void (*finalize) (void);
  void* lib1;
  void* lib2;
  void* lib3;
  if (rdrand_supported ())
    {
      lib1 = dlopen("randlibhw.so", RTLD_NOW);
      if (lib1 == NULL)
        {
       fprintf(stderr, "fail to open randlibhw.so.\n");
       return 1;
        }
```

```
     void (*hardware_rand64_init)(void) = dlsym(lib1,
"hardware_rand64_init");
     if (hardware_rand64_init != NULL)
       {
      initialize = hardware_rand64_init;
       }

     lib2 = dlopen("randlibhw.so", RTLD_NOW);
     if (lib2 == NULL)
       {
      fprintf(stderr, "fail to open randlibhw.so.\n");
      return 1;
       }
     unsigned long long (*hardware_rand64)(void) = dlsym(lib2,
"hardware_rand64");
     if (hardware_rand64 != NULL)
       {
      rand64 = hardware_rand64;
       }

     lib3 = dlopen("randlibhw.so", RTLD_NOW);
     if (lib3 == NULL)
       {
      fprintf(stderr, "fail to open randlibhw.so.\n");
      return 1;
       }
     void (*hardware_rand64_fini)(void) = dlsym(lib3,
"hardware_rand64_fini");
     if (hardware_rand64_fini != NULL)
       {
      finalize = hardware_rand64_fini;
       }
   }
  else
    {
     lib1 = dlopen("randlibsw.so", RTLD_NOW);
     if (lib1 == NULL)
       {
      fprintf(stderr, "fail to open randlibsw.so.\n");
      return 1;
       }
     void (*software_rand64_init)(void) = dlsym(lib1,
"software_rand64_init");
     if (software_rand64_init != NULL)
```

```
       {
       initialize = software_rand64_init;
       }

     lib2 = dlopen("randlibsw.so", RTLD_NOW);
     if (lib2 == NULL)
       {
       fprintf(stderr, "fail to open randlibsw.so.\n");
       return 1;
       }
     unsigned long long (*software_rand64)(void) = dlsym(lib2,
"software_rand64");
     if (software_rand64 != NULL)
       {
       rand64 = software_rand64;
       }

     lib3 = dlopen("randlibsw.so", RTLD_NOW);
     if (lib3 == NULL)
       {
       fprintf(stderr, "fail to open randlibsw.so.\n");
       return 1;
       }
     void (*software_rand64_fini)(void) = dlsym(lib3,
"software_rand64_fini");
     if (software_rand64_fini != NULL)
       {
       finalize = software_rand64_fini;
       }
   }

  initialize ();
  int wordsize = sizeof rand64 ();
  int output_errno = 0;

    do
      {
       unsigned long long x = rand64 ();
       size_t outbytes = nbytes < wordsize ? nbytes : wordsize;
       if (fwrite (&x, 1, outbytes, stdout) != outbytes)
       {
           output_errno = errno;
           break;
       }
```

```
        nbytes -= outbytes;
      }
    while (0 < nbytes);

    if (fclose (stdout) != 0)
      output_errno = errno;

    if (output_errno)
      {
        errno = output_errno;
        perror ("output");
        finalize ();
        return 1;
      }

    finalize ();
    dlclose(lib1);
    dlclose(lib2);
    dlclose(lib3);
    return 0;
}
```