Senni Tan 400196392

fib.c

#include "bmp.h"

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

/\*determine the lenth of the nth fibonacci word\*/

int fib\_size(int n){

if (n==0) return 0;

if ( n==1 || n==2 ) return 1;

return fib\_size(n-1) + fib\_size(n-2);

}

/\*make the fibonacci string\*/

char \* fibword(int n){

int len = fib\_size(n);

char \*word = malloc((len+1) \* sizeof(char));

char \*tmp = malloc((len+1) \* sizeof(char));

char \*Sn\_1 = malloc((len+1) \* sizeof(char));

char \*Sn = malloc((len+1) \* sizeof(char));

char one[2] = {'1'};

char zero[2] = {'0'};

char zero\_one[3] = {'0','1'};

if (n==0) return NULL;

if (n==1) {

strcpy(word,one);

free(tmp);

free(Sn);

free(Sn\_1);

return word;

}

if (n==2) {

strcpy(word,zero);

free(tmp);

free(Sn);

free(Sn\_1);

return word;

}

if (n==3) {

strcpy(word,zero\_one);

free(tmp);

free(Sn);

free(Sn\_1);

return word;

}

strcpy(Sn\_1,zero);

strcpy(Sn,zero\_one);

n-=1;

int i;

for (i = 3; i <= n; i++){

strcpy(tmp,Sn);

strcat(Sn,Sn\_1);

strcpy(Sn\_1, tmp);

}

strcpy(word, Sn);

free(tmp);

free(Sn);

free(Sn\_1);

return word;

}

/\*direction\*/

typedef enum {

up,

down,

left,

right

} direction;

/\*draw a segment\*/

void draw(int x, int y, direction dir, int step, int w, RGB\* im, RGB c){

#define I(i, j) im[(i)\*w + j]

int j;

if (dir == up)

for (j = 0; j < step; j++)

I(x + j, y) = c;

if (dir == down)

for (j = 0; j < step; j++)

I(x - j, y) = c;

if (dir == right)

for (j = 0; j < step; j++)

I(x, y + j) = c;

if (dir == left)

for (j = 0; j < step; j++)

I(x, y - j) = c;

#undef I

}

/\*make a turn in direction\*/

direction turn(direction dir, direction to\_turn90){

if (to\_turn90 == right){

if (dir == up){

return right;

}

if (dir == right){

return down;

}

if (dir == down){

return left;

}

if (dir == left){

return up;

}

}

if (to\_turn90 == left){

if (dir == up){

return left;

}

if (dir == left){

return down;

}

if (dir == down){

return right;

}

if (dir == right){

return up;

}

}

}

/\*determine if a num is even or not\*/

int is\_even(int i){

if ((i%2) == 0){

return 1;

}

if ((i%2) == 1){

return 0;

}

}

int fib(int n, int x, int y, int step, RGB b, RGB f, int w, int h, RGB\* image){

char \*fn = fibword(n);

if (!fn) return 0;

int i;

/\*set blackground\*/

for (i = 0; i < w \* h; i++)

image[i] = b;

int len = strlen(fn);

int dig = 0;

direction dir = up;

for (i = 1; i <= len && x < w && y < h; i++){

dig = fn[i-1] - '0';

draw(x, y, dir, step, w, image, f);

switch (dir){

case (up):

x += step;

break;

case (right):

y += step;

break;

case (down):

x -= step;

break;

case (left):

y -= step;

break;

default:

break;

}

if (dig == 0){

if (is\_even(i)){

dir = turn(dir, left);

}

else{

dir = turn(dir, right);

}

}

if (dig == 1){

dir = dir;

}

}

free(fn);

if ((i-1) != len)

return 0;

return len;

}

makefile.b

CFLAGS=-Wall -O2 -ansi

fib: main\_fib.o fib.o bmp.o timing.o

$(CXX) -o fib $?

runall:

./fib 7 10 10 10 100 100 fib7.bmp

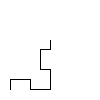
./fib 9 10 10 10 300 300 fib9.bmp

./fib 25 10 10 10 10000 10000 fib25.bmp

./fib 26 10 10 10 20000 20000 fib26.bmp

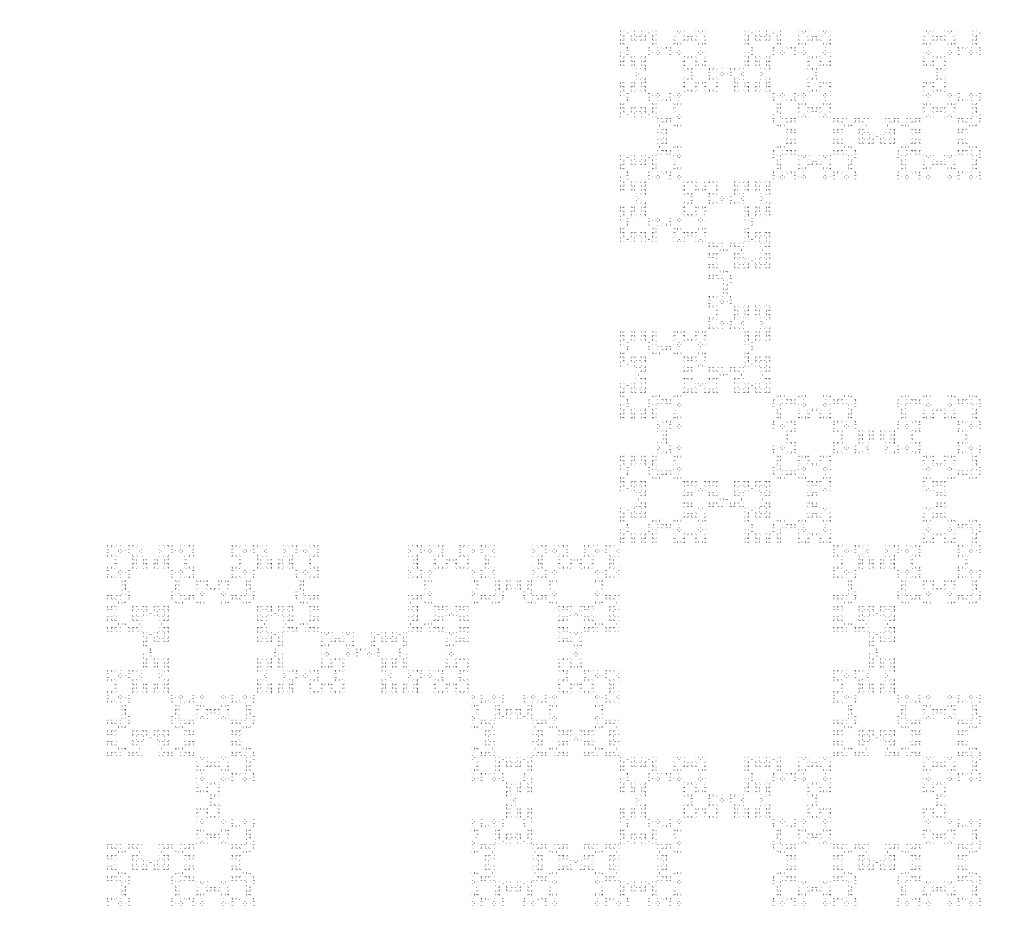
clean:

@rm -rf fib \*.o \*bmp

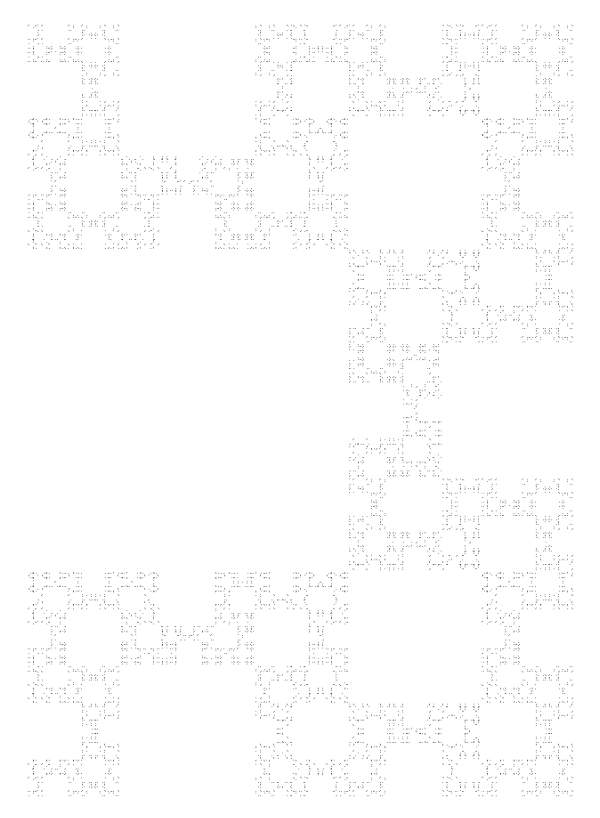
n = 7

A close up of a logo

Description automatically generated n = 9



n = 25



n = 26

Problem 2

Flat profile:

Each sample counts as 0.01 seconds.

% cumulative self self total

time seconds seconds calls ms/call ms/call name

17.98 2.30 2.30 fib\_size (fib.c:9 @ 400bc2)

12.66 3.93 1.62 saveBMP (bmp.c:25 @ 4011ab)

10.20 5.24 1.31 fib\_size (fib.c:11 @ 400b98)

8.60 6.34 1.10 draw (fib.c:71 @ 400bf2)

5.94 7.10 0.76 draw (fib.c:87 @ 400c24)

5.63 7.82 0.72 fib\_size (fib.c:11 @ 400bb8)

4.61 8.41 0.59 1 591.26 591.26 draw (fib.c:71 @ 400bd0)

4.30 8.96 0.55 saveBMP (bmp.c:15 @ 4011a3)

4.30 9.52 0.55 saveBMP (bmp.c:27 @ 4011bd)

3.67 9.99 0.47 saveBMP (bmp.c:26 @ 4011ae)

3.20 10.40 0.41 fib\_size (fib.c:10 @ 400b88)

2.89 10.77 0.37 draw (fib.c:75 @ 400c1d)

2.50 11.09 0.32 saveBMP (bmp.c:14 @ 40118f)

1.60 11.29 0.21 saveBMP (bmp.c:15 @ 40119c)

1.49 11.48 0.19 draw (fib.c:75 @ 400c10)

1.49 11.67 0.19 draw (fib.c:76 @ 400c13)

1.49 11.87 0.19 saveBMP (bmp.c:25 @ 4011a0)

1.41 12.05 0.18 saveBMP (bmp.c:31 @ 4011b5)

1.37 12.22 0.18 fib\_size (fib.c:8 @ 400b92)

1.25 12.38 0.16 draw (fib.c:74 @ 400bec)

1.25 12.54 0.16 draw (fib.c:75 @ 400bee)

0.55 12.61 0.07 fib\_size (fib.c:9 @ 400bb4)

0.31 12.65 0.04 draw (fib.c:77 @ 400c30)

0.23 12.68 0.03 fib (fib.c:135 @ 400f4f)

0.23 12.71 0.03 fib (fib.c:141 @ 400f8d)

0.16 12.73 0.02 fib (fib.c:131 @ 400f45)

0.12 12.75 0.02 saveBMP (bmp.c:25 @ 40119a)

0.08 12.76 0.01 fib (fib.c:138 @ 400f7f)

0.08 12.77 0.01 fib (fib.c:141 @ 400fc9)

0.08 12.78 0.01 fib (fib.c:144 @ 401019)

0.08 12.79 0.01 fib (fib.c:141 @ 40105f)

0.08 12.80 0.01 saveBMP (bmp.c:26 @ 4011ba)

0.08 12.81 0.01 saveBMP (bmp.c:43 @ 4012a6)

0.04 12.81 0.01 fib (fib.c:141 @ 401059)

0.04 12.82 0.01 fib (fib.c:162 @ 40105c)

0.04 12.82 0.01 fib (fib.c:172 @ 40106c)

0.04 12.83 0.01 fib (fib.c:173 @ 401075)

0.00 12.83 0.00 1346269 0.00 0.00 fib (fib.c:130 @ 400f00)

0.00 12.83 0.00 514230 0.00 0.00 saveBMP (bmp.c:6 @ 4010e0)

0.00 12.83 0.00 3 0.00 0.00 main (main\_fib.c:9 @ 400890)

0.00 12.83 0.00 1 0.00 0.00 turn (fib.c:89 @ 400d00)

% the percentage of the total running time of the

time program used by this function.

cumulative a running sum of the number of seconds accounted

seconds for by this function and those listed above it.

self the number of seconds accounted for by this

seconds function alone. This is the major sort for this

listing.

calls the number of times this function was invoked, if

this function is profiled, else blank.

self the average number of milliseconds spent in this

ms/call function per call, if this function is profiled,

else blank.

total the average number of milliseconds spent in this

ms/call function and its descendents per call, if this

function is profiled, else blank.

name the name of the function. This is the minor sort

for this listing. The index shows the location of

the function in the gprof listing. If the index is

in parenthesis it shows where it would appear in

the gprof listing if it were to be printed.

Call graph (explanation follows)

granularity: each sample hit covers 2 byte(s) for 0.08% of 12.83 seconds

index % time self children called name

0.59 0.00 1/1 turn (fib.c:108 @ 400d1c) [8]

[7] 4.6 0.59 0.00 1 draw (fib.c:71 @ 400bd0) [7]

-----------------------------------------------

0.00 0.00 257114/1346269 saveBMP (bmp.c:42 @ 4012a0) [261]

0.00 0.00 257115/1346269 saveBMP (bmp.c:40 @ 401290) [260]

0.00 0.00 832040/1346269 saveBMP (bmp.c:39 @ 401224) [256]

[39] 0.0 0.00 0.00 1346269 fib (fib.c:130 @ 400f00) [39]

-----------------------------------------------

0.00 0.00 1/514230 \_fini [287]

0.00 0.00 514229/514230 saveBMP (bmp.c:5 @ 401269) [259]

[40] 0.0 0.00 0.00 514230 saveBMP (bmp.c:6 @ 4010e0) [40]

-----------------------------------------------

0.00 0.00 1/3 \_\_do\_global\_ctors\_aux [283]

0.00 0.00 2/3 \_fini [287]

[41] 0.0 0.00 0.00 3 main (main\_fib.c:9 @ 400890) [41]

-----------------------------------------------

0.00 0.00 1/1 saveBMP (bmp.c:23 @ 401170) [238]

[42] 0.0 0.00 0.00 1 turn (fib.c:89 @ 400d00) [42]

-----------------------------------------------

This table describes the call tree of the program, and was sorted by

the total amount of time spent in each function and its children.

Each entry in this table consists of several lines. The line with the

index number at the left hand margin lists the current function.

The lines above it list the functions that called this function,

and the lines below it list the functions this one called.

This line lists:

index A unique number given to each element of the table.

Index numbers are sorted numerically.

The index number is printed next to every function name so

it is easier to look up where the function in the table.

% time This is the percentage of the `total' time that was spent

in this function and its children. Note that due to

different viewpoints, functions excluded by options, etc,

these numbers will NOT add up to 100%.

self This is the total amount of time spent in this function.

children This is the total amount of time propagated into this

function by its children.

called This is the number of times the function was called.

If the function called itself recursively, the number

only includes non-recursive calls, and is followed by

a `+' and the number of recursive calls.

name The name of the current function. The index number is

printed after it. If the function is a member of a

cycle, the cycle number is printed between the

function's name and the index number.

For the function's parents, the fields have the following meanings:

self This is the amount of time that was propagated directly

from the function into this parent.

children This is the amount of time that was propagated from

the function's children into this parent.

called This is the number of times this parent called the

function `/' the total number of times the function

was called. Recursive calls to the function are not

included in the number after the `/'.

name This is the name of the parent. The parent's index

number is printed after it. If the parent is a

member of a cycle, the cycle number is printed between

the name and the index number.

If the parents of the function cannot be determined, the word

`<spontaneous>' is printed in the `name' field, and all the other

fields are blank.

For the function's children, the fields have the following meanings:

self This is the amount of time that was propagated directly

from the child into the function.

children This is the amount of time that was propagated from the

child's children to the function.

called This is the number of times the function called

this child `/' the total number of times the child

was called. Recursive calls by the child are not

listed in the number after the `/'.

name This is the name of the child. The child's index

number is printed after it. If the child is a

member of a cycle, the cycle number is printed

between the name and the index number.

If there are any cycles (circles) in the call graph, there is an

entry for the cycle-as-a-whole. This entry shows who called the

cycle (as parents) and the members of the cycle (as children.)

The `+' recursive calls entry shows the number of function calls that

were internal to the cycle, and the calls entry for each member shows,

for that member, how many times it was called from other members of

the cycle.

Index by function name

[7] draw (fib.c:71 @ 400bd0) [30] fib (fib.c:141 @ 400fc9) [40] saveBMP (bmp.c:6 @ 4010e0)

[21] draw (fib.c:74 @ 400bec) [31] fib (fib.c:144 @ 401019) [14] saveBMP (bmp.c:14 @ 40118f)

[22] draw (fib.c:75 @ 400bee) [35] fib (fib.c:141 @ 401059) [28] saveBMP (bmp.c:25 @ 40119a)

[4] draw (fib.c:71 @ 400bf2) [36] fib (fib.c:162 @ 40105c) [15] saveBMP (bmp.c:15 @ 40119c)

[16] draw (fib.c:75 @ 400c10) [32] fib (fib.c:141 @ 40105f) [18] saveBMP (bmp.c:25 @ 4011a0)

[17] draw (fib.c:76 @ 400c13) [37] fib (fib.c:172 @ 40106c) [9] saveBMP (bmp.c:15 @ 4011a3)

[13] draw (fib.c:75 @ 400c1d) [38] fib (fib.c:173 @ 401075) [2] saveBMP (bmp.c:25 @ 4011ab)

[5] draw (fib.c:87 @ 400c24) [12] fib\_size (fib.c:10 @ 400b88) [11] saveBMP (bmp.c:26 @ 4011ae)

[24] draw (fib.c:77 @ 400c30) [20] fib\_size (fib.c:8 @ 400b92) [19] saveBMP (bmp.c:31 @ 4011b5)

[39] fib (fib.c:130 @ 400f00) [3] fib\_size (fib.c:11 @ 400b98) [33] saveBMP (bmp.c:26 @ 4011ba)

[27] fib (fib.c:131 @ 400f45) [23] fib\_size (fib.c:9 @ 400bb4) [10] saveBMP (bmp.c:27 @ 4011bd)

[25] fib (fib.c:135 @ 400f4f) [6] fib\_size (fib.c:11 @ 400bb8) [34] saveBMP (bmp.c:43 @ 4012a6)

[29] fib (fib.c:138 @ 400f7f) [1] fib\_size (fib.c:9 @ 400bc2) [42] turn (fib.c:89 @ 400d00)

[26] fib (fib.c:141 @ 400f8d) [41] main (main\_fib.c:9 @ 400890)

**The first line of my fib function takes most of the time. The first line in fib function calls a subfunction “fib\_size” which calculate the nth Fibonacci word’s length with recursion**. **The recursion waits for the stack of the base case return back, that’s why it takes the most of the time.**

filter.c

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

typedef struct pixel {

unsigned char r, g, b;

} Pixel;

typedef struct image {

unsigned int width, height, max;

Pixel \*\*data;

} Image;

void printPPM(Image \*image) {

int i,j;

for (i=0; i< image->height; i++){

for (j=0; j< image->width; j++){

printf("#%02x%02x%02x", image->data[i][j].r, image->data[i][j].g, image->data[i][j].b);

}

printf("\n");

}

}

Image \*readPPM(char \*file\_name) {

FILE \*file = fopen(file\_name, "r");

if (!file) {

fprintf(stderr, "Unable to open file \"%s\"\n", file\_name);

return NULL;

}

char format[3];

/\*fscanf(file, "%2s\n", format);\*/

if(fscanf(file, "%s\n", format) != 1)

return NULL;

if (strcmp(format, "P3"))

return NULL;

Image \*image = malloc(sizeof(Image));

if(fscanf(file, "%u %u %u", &image->width, &image->height, &image->max) != 3)

return NULL;

image->data = malloc(sizeof(Pixel \*) \* image->height);

int i, j;

for (i = 0; i < image->height; i++)

image->data[i] = malloc(sizeof(Pixel) \* image->width);

for (i = 0; i < image->height; i++)

for (j = 0; j < image->width; j++){

int pixels\_read = fscanf(file, "%hhu %hhu %hhu", &(image->data[i][j].r), &(image->data[i][j].g), &(image->data[i][j].b));

if (pixels\_read != 3)

return NULL;

}

fclose(file);

return image;

}

int writePPM(char \*file\_name, Image \*image){

FILE \*file = fopen(file\_name, "w");

if (!file) {

fprintf(stderr, "Unable to open file \"%s\"\n", file\_name);

return -1;

}

fprintf(file, "P3\n");

fprintf(file, "%u %u\n", image->width, image->height);

fprintf(file,"%u\n", image->max);

int i,j;

for (i = 0; i < image->height; i++)

{

for (j = 0; j < image->width; j++)

{

fprintf(file, "%u %u %u ", image->data[i][j].r, image->data[i][j].g, image->data[i][j].b);

}

fprintf(file, "\n");

}

fclose(file);

return 0;

}

void filter(Image \*input, Image \*output, int \*kernel, int n, int scale){

#define KK(k,l) kernel[(k) \* n + (l)]

int i,j,k,l;

int red, green, blue;

int x, y;

output->data = malloc(sizeof(Pixel\*) \* output->height);

for (i=0; i<output->height; i++){

output->data[i] = malloc(sizeof(Pixel) \* output->width);

}

i = 0;

while (i < input->height){

j = 0;

while (j < input->width)

{

red = 0;

green = 0;

blue = 0;

k = 0;

while (k < n)

{

l = 0;

while (l < n)

{

x = i + k - (n/2);

y = j + l - (n/2);

if (x>=0 && x<input->height && y>=0 && y<input->width){

red += ((int)((input->data)[x][y].r) \* (KK(k,l))) / scale;

green += ((int)((input->data)[x][y].g) \* (KK(k,l))) / scale;

blue += ((int)((input->data)[x][y].b) \* (KK(k,l))) / scale;

}

else{

red += 0;

green += 0;

blue += 0;

}

l++;

}

k++;

}

if (red < 0)

red = 0;

else if (red > 255)

red = 255;

if (green < 0)

green = 0;

else if (green > 255)

green = 255;

if (blue < 0)

blue = 0;

else if (blue > 255)

blue = 255;

output->data[i][j].r = (unsigned char)red;

output->data[i][j].g = (unsigned char)green;

output->data[i][j].b = (unsigned char)blue;

j++;

}

i++;

}

#undef KK

}

int main(int argc, char \*\*argv){

if (argc != 4){

printf("Usage: ./filter input\_pic.ppm kernel output\_pic.ppm\n");

return -1;

}

char \* input = argv[1];

if (!input) {

printf("Can not open the input ppm file\n");

return -1;

}

char \* kernel = argv[2];

if (!kernel) {

printf("Can not open the Kernel file\n");

return -1;

}

char \* output = argv[3];

if (!output){

printf("Problem with the output ppm file\n");

return -1;

}

FILE \*f;

f = fopen(kernel, "r");

int size, scale;

if(fscanf(f,"%d",&size)!= 1)

return 0;

if(fscanf(f,"%d",&scale) != 1)

return 0;

int \* K = malloc(sizeof(int) \* size \* size);

int i;

for(i = size\*size - 1; i >= 0; i--){

if(fscanf(f, "%d", K+i) != 1){

return 0;

}

}

fclose(f);

Image \*in\_img = readPPM(input);

Image \*out\_img =(Image\*)malloc(sizeof(Image));

out\_img->width = in\_img->width;

out\_img->height = in\_img->height;

out\_img->max = in\_img->max;

filter(in\_img,out\_img,K,size,scale);

writePPM(output,out\_img);

return 0;

}

makefile.f

CFLAGS=-Wall -O2 -ansi

filter: filter.o

$(CXX) -o filter $?

clean :

@rm -rf filter \*.o \*.ppm

Cities.c

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

#include <malloc.h>

typedef struct

{

char \* city;

char \* country;

double population;

}City;

City cities[10005];

int cityn = 0;

int compare(const void \* a, const void \* b) {

City \* Ca = (City \*)a;

City \* Cb = (City \*)b;

return Cb->population - Ca->population;

}

int main()

{

FILE\* fp = fopen("cities.csv", "r");

char line[1005];

int linen = 0;

int i;

while (fgets(line, 1005, fp))

{

if (line[strlen(line) - 1] == '\n')

line[strlen(line) - 1] = '\0';

if (linen > 0)

{

char delims[] = ",";

char\* result = NULL;

char\* record = NULL;

record = strtok(line, ",");

cities[cityn].city = (char\*)malloc(sizeof(char) \* 1005);

strcpy(cities[cityn].city, record);

record = strtok(NULL, ",");

record = strtok(NULL, ",");

record = strtok(NULL, ",");

record = strtok(NULL, ",");

cities[cityn].population = atof(record);

record = strtok(NULL, ",");

cities[cityn].country = (char\*)malloc(sizeof(char) \* 1005);

strcpy(cities[cityn].country, record);

cityn++;

}

linen++;

}

qsort(cities, cityn, sizeof(City), compare);

fp = fopen("sorted.csv", "w");

for (i = 0; i < cityn;i++)

{

fprintf(fp, "%s,%d,%s\n", cities[i].city, (int)cities[i].population, cities[i].country);

}

fclose(fp);

}