**Honors Computer Science I**

**Exam #2**

**Name: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

1) Recurrence Relations – 15 pts

Use the iteration technique to find a solution to the following recurrence relation:

2) Base Conversion – 10 pts

a) Convert 1943 in base 10 to base 7.

b) Convert 3EF86C16 to octal (base 8).

3) Timing Problems – 10 pts

a) For an input size of n = 40,000 an O(n2) algorithm takes 8 seconds to complete. How long should we expect the same algorithm to take for an input size of n = 70000?

b) It is known that the run time of a particular algorithm is O(nk) for some constant k. For an input size of n = 30,000, the algorithm takes 24 microseconds. For an input size if 270,000, the algorithm takes 72 microseconds. What is the most likely value of k?

4) Summations – 5 pts

Find the value of the following sum, in terms of n: .

5) Stack Question – 10 pts

Convert the following infix expression to postfix. Please show the state of the operator stack at each of the indicated points (A, B, C).

A B C

( 3 + 7 – 2 ) \* 6 – 2 \* ( 4 / ( 3 – 2 ) \* ( 5 – 3 ) + 2 )

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

A B C

Resulting postfix expression:

6) Queue Floodfill – 25 pts

As discussed in class, a floodfill written recursively on a large grid (say 1000 x 1000) may crash. To avoid this, the floodfill can be rewritten iteratively, using a queue. The idea is as follows: start at the initial square, adding it to the queue. Then, while the queue isn’t empty, dequeue the next item, then enqueue all of its unvisited neighbors that are supposed to be filled, marking those neighbors (so they don’t get enqueued multiple times). Assume you have access to call the following functions:

void init(struct queue\* q); // initializes q to be empty

void enqueue(struct queue\* q, int value); // enqueues value to q

int dequeue(struct queue\* q); // returns the item dequeued.

int empty(struct queue\* q); // returns 1 iff q is empty.

const int N = 1000;

const int FILLVALUE = -99;

Complete the function on the following page that takes in a 2D array (int\*\*) of a N x N grid (where N is a constant defined above), as well as a x and y coordinate, each in [0, 999], and fills each square that currently contains the original value of grid[x][y] with -99 (FILLVALUE), that is in the contiguous area storing the same value as grid[x][y]. Perform the floodfill by only permitting adjacent squares that are directly above, below, to the left, or to the right of previously filled squares. For example, if a portion of the grid contains what is shown on the left before the floodfill, it should look like what is shown on the right, afterwards, if our initial fill square was (1,1):

2 3 4 2 2 4 2 3 4 -99 -99 4

1 2 2 2 2 3 1 -99 -99 -99 -99 3

3 2 3 6 2 2 3 -99 3 6 -99 -99

Assume you also have access to the following inbounds function:

int inbounds(int x, int y) {

return x >= 0 && x < N && y >= 0 && y < N;

}

Note: Finally, you must think about how you would store an (x,y) coordinate as one single integer, since our queue only stores integers.

void floodfill(int\*\* grid, int x, int y) {

int fillVal = grid[x][y];

struct queue list;

init( \_\_\_\_\_\_\_\_\_ );

int\* used = calloc( \_\_\_\_\_\_\_\_ , sizeof(int));

enqueue(&list, \_\_\_\_\_\_\_\_ );

used[N\*x+y] = \_\_\_\_ ;

while (!empty(list)) {

int next = \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ ;

int curx = \_\_\_\_\_\_\_\_\_\_\_\_\_ ;

int cury = \_\_\_\_\_\_\_\_\_\_\_\_\_ ;

grid[curx][cury] = \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ ;

if (inbounds(curx+1,cury) && !used[(curx+1)\*N+cury] &&

grid[curx+1][cury] == \_\_\_\_\_\_\_\_\_) {

enqueue(&list, (curx+1)\*N+cury);

used[(curx+1)\*N+cury] = 1;

}

if (inbounds(curx-1,cury) && !used[(curx-1)\*N+cury] &&

grid[curx-1][cury] == \_\_\_\_\_\_\_\_\_) {

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ ;

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ ;

}

if (inbounds(curx,cury+1) && !used[curx\*N+cury+1] &&

grid[curx][cury+1] == \_\_\_\_\_\_\_\_\_) {

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ ;

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ ;

}

if (inbounds(curx,cury-1) && !used[curx\*N+cury-1] &&

grid[curx][cury-1] == \_\_\_\_\_\_\_\_\_) {

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ ;

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ ;

}

}

free(used);

}

7) Jumble Case Study – 24 pts

a) A Jumble is where a set of scrambled letters is given a person and they have to figure out how to rearrange the letters to spell a word. One method of solution would be to try each permutation of the letters and then search for each of these permutations in the dictionary. Given a sorted dictionary of size D and a list of N letters to unscramble, what is the run-time of using this algorithm, in terms of D and N? Give a brief justification for your answer.

b) A second approach to solving the problem would be to take the dictionary and store each word itself in sorted order. For example, the word “PHONE” would be stored as “EHNOP”. Then, store both strings (sorted and original word) in a single struct. Finally, sort this struct by the “sorted” version of the word. To solve the jumble, take the input scrambled letters, sort them, and then search the modified dictionary with this value. The output would be each index in the array with a matching entry. What is the run-time of using this algorithm, in terms of D and N? (These variables represent the same items as in part a.)

c) A third approach to solve the jumble problem would be to use backtracking. Try each letter in each slot, but if the beginning set of letters you are trying can’t form a word, stop the search. For example, given the letters BRLMEPO, we can skip all permutations that start with BRL because no valid word starts with BRL. (We can’t say the same for BR, though.) Doing this will speed up our solution compared to the approach given in part a. Assume you have access to a function with the following specification:

// Returns 0 if no valid words start with the substring

// str[0..k-1]. Returns 1 otherwise.

int canstart(char\* str, int k);

Note: This function can verify whether or not a whole string is a word by passing in strlen(str) as the second parameter.

Complete the function printAllRec on the next page so that a call to printAllJumbles prints out all solutions to a Jumble puzzle. (Note: assume the input string has no repeated characters.)

void printAllJumbles(char\* mixed) {

printAllRec(mixed, 0);

}

void swap(char\* word, int i, int j) {

char temp = word[i];

word[i] = word[j];

word[j] = temp;

}

// Prints all valid rearrangements of mixed that form words

// with the first k characters fixed.

void printAllRec(char\* mixed, int k) {

}

8) (1 pt) Who is the host of the show, Anderson Cooper 360°? \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Scratch Page – Please clearly mark the work on this page you would like graded.**