Foundation Exam - Sample Problems for New Topics

Fall 2016

Dynamic Memory Allocation

Sample Question #1

Consider the following struct to store a single Uno playing card:

```
typedef struct {
   char color[20];
   int number;
} UnoCard;
```

Write a function that takes in a single string, *mycolor*, and a single positive integer, *quantity*, and returns a pointer to an array of *quantity* UnoCard structs. Each of these structs should be assigned to have the color *mycolor* and each should be assigned the units digit of the index in which it's stored. (Thus, the card stored at index 27 should store the number 7, for example.) You may assume that string.h is included.

Sample Question #2

Solve a similar question to #1, but this time have the function return a pointer to an array of pointers to type UnoCard, where each of these pointers points to a ***single*** UnoCard struct. Store the same information as delineated in question #1 in each struct.

Sample Question #3

Assume that the pointer *cards* points to the structure allocated by the function in sample question #2. Assume that the variable *quantity* stores the length of the array that the pointer *cards* points to. Write a segment of code to free all the associated memory.

Sample Question #4

Write a segment of code that reads in a positive integer into the variable *n*, and then allocates an array of arrays of type int. The length of the array stored at index *i* should be length *i*+1. Leave all of the *n* arrays uninitialized.

Continued on the following page...

Tries

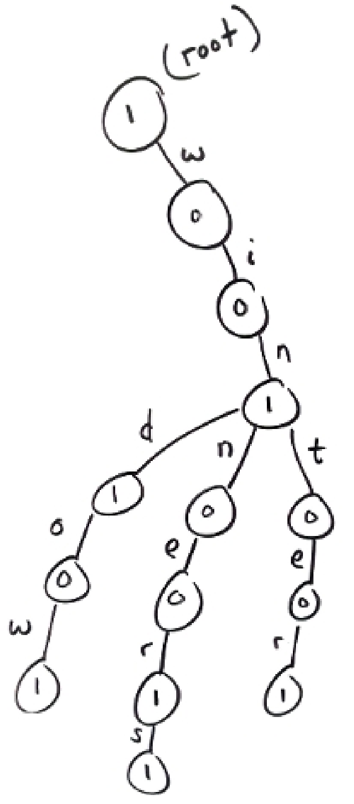
Sample Question #1

(a) What strings are represented in the following trie?

(b) Show what the trie looks like after inserting the strings "winnowing" and "virtue".

(c) Show what the trie looks like after deleting each of the following strings: "window", "winners", and "win".

(d) How many unique two-letter strings could you insert into the following trie without creating any new nodes? (By "unique", I mean that you should not account for the insertion of multiple instances of some string.) Assume we are restricted to strings that contain only lowercase alphabetic characters.



Continued on the following page...

Sample Question #2

Write an **iterative** function that takes the root of a trie and some string, *str*, and inserts that string into the trie. The function signature is *TrieNode *insert(TrieNode *root, char *str)*, and the TrieNode struct definition is as follows:

```
typedef struct TrieNode
{
   // Children nodes.
   struct TrieNode *children[26];

   // Number of times this string occurs in the trie.
   int count;
} TrieNode;
```

The function should return the root of your trie. Note that the root passed to your function might be NULL, in which case you should return the new root that you create for the trie.

You may assume the string passed to the function is non-NULL and non-empty, and contains only alphabetic characters. However, you must make sure your function is case insensitive. For example, insert("SoMeSTriNG") should insert "somestring" into the trie.

Sample Question #3

Write a **recursive** version of the function from the previous question.

Continued on the following page...

Bitwise Operators (and Two's Complement)

Sample Question #1

What are the values of the following expressions in C?

a. `37 & 44`

b. `15 | 27`

c. `26 ^ 17`

d. `111 >> 3`

e. `3 << 4`

f. `(13 & (1 << 3)) >> 3`

g. `(11>>2) & 1`

Sample Question #2

What is the output of the following segment of code?

```
int i;
for (i=0; i<16; i++) {
    int res = 0, j = 0;
    for (j=0; (1<<j) <= i; j++) {
        if ((i & (1<<j)) != 0) res++;
    }
    printf("%d ", res);
}
```

Sample Question #3

Assuming we're using two's complement, what base 10 integer values do the following 32-bit binary numbers represent?

a. `10000000000000000000000000000000`

b. `11111111111111111111111111111111`

c. `10000000000000000000000000000001`

d. `01000000000000000000000000000000`

e. `11000000000000000000000000000000`

f. `11111111111111111111111111010101`

g. `01111111111111111111111111010101`

Sample Question #4

For each of the 32-bit strings in the previous question, give a few lines of code that would result in an integer variable having that underlying binary representation. (Don't just calculate the base 10 integer by hand and then assign it to an integer variable. Reason your way through this using bitwise operators.) Assume the only integer values you can hard-code are -1 through 32 and INT_MIN, INT_MAX, UINT_MIN, and UINT_MAX. You must use bitwise operators in your solutions, and you must not use the *pow()* function at all.

For example, we could create an integer whose underlying binary representation matches the one from part (c) with the following lines of code:

```
int i = 1;      // now we have 00000000000000000000000000000001
i = i << 31;    // now we have 10000000000000000000000000000000
i = i | 1;      // now we have 10000000000000000000000000000001
```

An alternative solution would be:

```
int i = INT_MAX;     // now we have 01111111111111111111111111111111
i = ~i;              // now we have 10000000000000000000000000000000
i = i ^ 1;           // now we have 10000000000000000000000000000001
```

Sample Question #5

For each of the following lines of code, give the 32-bit binary representation underlying the variable *i*. You may assume we're using two's complement.

```
int i = INT_MAX;
i = ~i;
i = i ^ 1;
```

Sample Question #6

What integer value would be printed by each of the following lines of code? (Hint: Start by converting each of the integers to binary by hand, and then convert the result back to decimal.)

```
a. printf("%d\n", 33 & 51);

b. printf("%d\n", 33 ^ 51);

c. printf("%d\n", 33 | 51);

d. printf("%d\n", ~33);

e. printf("%d\n", ~(-33));
```

Continued on the following page...

Sample Question #7

Test your understanding of two's complement and bitwise operators: What values will be printed by the following code?

```
#include <stdio.h>

int main(void)
{
   int beast = 1;

   printf("%d\n", beast << 1);
   printf("%d\n", beast);
   printf("%d\n", beast << 2);
   printf("%d\n", beast << 3);
   printf("%d\n", beast << 4);
   printf("%d\n", 108 << 3);
   printf("%d\n", beast >> 2);
   beast = beast << 31;
   printf("%d\n", beast);
   beast = ~beast;
   printf("%d\n", beast);
   printf("%u\n", beast);
   beast = ~1;
   printf("%d\n", beast);
   beast = ~0;
   printf("%d\n", beast);
   printf("%u\n", beast);
   printf("%d\n", ~75);
   printf("%d\n", ~78);
   printf("%d\n", 0xBEEF);
   printf("%d\n", 020);

   return 0;
}
```

Sample Question #8

What will be the output of the following program? How would the program's output differ if *beast* were a signed integer (i.e., *int beast* instead of *unsigned int beast*), assuming the *printf()* statements were modified accordingly?

```
#include <stdio.h>

int main(void)
{
   unsigned int beast = 1;
   beast = beast << 31;
   printf("%u\n", beast);
   beast = beast >> 31;
   printf("%u\n", beast);
   return 0;
}
```

Continued on the following page...

Sample Question #9

What will be the output of the following lines of code?

```
printf("%x\n", 0xBEEF);
printf("%X\n", 0xBEEF);
printf("%d\n", 0xBEEF);
printf("%d\n", 0x255);
printf("%x\n", 255);
printf("%d\n", 52 & 39);
```

Continued on the following page...

Backtracking

Sample Question #1

An *s*-separated number of *n* digits is one where each pair of consecutive digits in the number has a difference (absolute value) of at least *s*. For example, 362958 is a 6 digit number that is 3-separated. (It's also 1- and 2-separated.) Write a function that prints out all positive integers of *n* digits that are *s*-separated. Your function should take in *n* and *s*, as well as two other values:

1. *curNum* – the current number being built.
2. *k* – the number of digits in curNum.

Here is the function prototype:

```
void printSepNums(int n, int s, int curNum, int k);
```

Sample Question #2

When attempting to place 4 queens on a 4 x 4 chessboard where no two can attack each other, a permutation solution tries 24 possible placements of queens, each with exactly 1 queen in each row. In the backtracking solution, certain permutations never get "tried" because they have a prefix that contains attacking queens. (For the purposes of this question, the permutation 1, 3, 2, 4 refers to placing queens on the coordinates (1, 1), (2, 3), (3, 2), and (4, 4), where the first value in the ordered pair is the row, the second the column.) List all pairs of values for the starting of a permutation that the backtracking solution attempts. (For example, 1, 2 doesn't get attempted because queens at (1, 1) and (2, 2) attack each other.)

Sample Question #3

The N-Queens problem asks, "How many ways are there to position exactly N queens on an NxN chess board such that none of the queens can attack one another?" The following backtracking function solves that problem. It places queens on a board, one per column, and each possible configuration that contains N queens that cannot attack each other gets printed exactly once.

Modify the backtracking function so that it prints only the first solution it finds and then terminates the backtracking process. The function should return all the way back to *nqueens()* without exploring any additional branches or printing any other board configurations after finding the first viable solution. The function should return 1 if it finds a solution, 0 if it finds no solutions.

```
int nqueens(int n)
{
   int retval;

   // The whichRow array stores which row is occupied by
   // the queen for each particular column. For example,
   // to place a queen in column 2, row 3, we would use:
   //
   // whichRow[2] = 3;

   int *whichRow = malloc(sizeof(int) * n);
```

```
      retval = backtracking(whichRow, n, 0);

      free(whichRow);

      return retval;
   }

   int backtrack(int *whichRow, int n, int col)
   {
      int row, total = 0;

      // Check whether we've filled all the columns.
      if (col == n)
      {
         // Assume this prints the board to the screen.
         printBoard(whichRow, n);
         return 1;
      }

      // Within this column, we will try placing the queen in each
      // possible row.
      for (row = 0; row < n; row++)
      {
         // Place the queen for this col at this particular row.
         whichRow[col] = row;

         // If placing the queen here is legal (i.e., the queen cannot
         // attack any other queens already placed on the board, then
         // make a recursive call where we move on to the next column.
         // Assume boardIsLegal() is properly defined elsewhere.
         if (boardIsLegal(whichRow, n, col))
            total += backtrack(whichRow, n, col + 1);
      }

      return total;
   }
```

Sample Question #4

Here is a puzzle for you (paraphrased from Wikipedia): A farmer goes to a market and purchases a fox, a goose, and a bag of beans. On his way home, the farmer approaches a river where there is a boat he can use to cross to the other side. However, the boat only has room for the farmer and exactly one of his purchases – the fox, the goose, or the bag of beans.

The farmer can't leave the goose alone with the bag of beans, or else the goose will eat the beans. The farmer can't leave the fox alone with the goose, or else the fox will eat the goose. (So, for example, if the farmer starts by ferrying the bag of beans across the river and drops it off there, then returns to the side where he left the fox and the goose, he'll find that the fox has eaten the goose. Goodbye, goose.)

The central question to this puzzle is: What steps must the farmer take to get all his purchases safely to the other side? (You will derive that solution in part (c) of this problem, below.)

* * *

Briefly explain how you would represent the different states of this problem in memory if you were tasked with implemented a backtracking solution that uses as few bytes as possible to represent a state. (I.e., what kind of data type(s) and/or data structure(s) would you use? Is there a technique or data structure we covered in Computer Science 1 that lends itself to a memory-efficient representation here?)

For example, one state we have to represent, the initial state of the problem, has the farmer, fox, goose, and bag of beans on one side of the river. Another state might be that the farmer is on one side of the river with a bag of beans, and the the goose and fox are left on the other side of the river. (Of course, that’s a bad state to be in, because the fox will eat the goose if the farmer isn’t there to stop it.)

Keep in mind that we also need a way to efficiently check off every state we’ve processed as we try to solve the problem, so we don’t get stuck in an infinite loop, processing the same states over and over again without making any actual progress toward the final goal. Briefly explain how you could keep track of which states have been processed already, given the representation you’ve chosen for the different states in this problem. (You don’t need to write any code; just give a brief, high-level overview of the idea behind your approach.)