COS 126	General Computer Science	Fall 1997

Final - Solutions[1]

Question 1

This is a FIFO queue, so we follow a first in, first out policy: E, EA, A, AS, S, SY, SYQ, SYQU, YQU, QU, QUE, QUES, QUEST, UEST, UESTI, ESTI, ESTIO, STIO, TIO, IO, O, ON, N, empty.

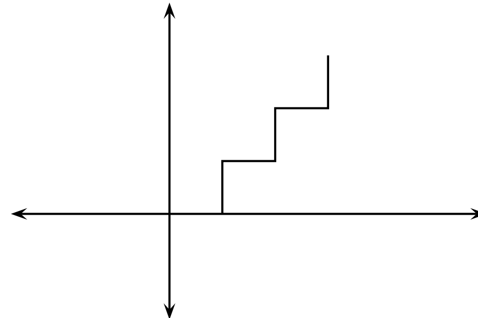
If this were a stack, we would follow a LIFO or last in, first out policy: E, AE, E, SE, E, YE, QYE, UQYE, QYE, YE, SYE, TSYE, SYE, OSYE, SYE, YE, E, empty, N, empty, crash.

Question 2

PostScript works like a stack. It has a number of primitive "turtle graphics" commands. We imagine that we have an obedient turtle that will follow our commands. The turtle also has a pen and will write as it moves, if directed to do so.

The `moveto` command pops two elements off the stack, say x and y, and the turtle moves to location (x, y), but without writing anything. The `rlineto` command pops two elements off the stack, say u and v, and the turtle moves in that direction (with the pen down) *relative* to its current location, say (x, y). So, the turtle moves from (x, y) to $(x + u, y + v)$ with the pen down. In contrast, if it were `lineto` instead of `rlineto`, the turtle would move from (x, y) to (u, v) with the pen down.

So, the command `100 0 moveto` causes the turtle to move to (100,0). The next command `0 100 rlineto` causes the turtle to move to (100,100) with the pen down. The next four `rlineto` commands cause the turtle to move from (100,100) to (200,100) to (200,200) to (300,200) and finally to (300,300), all with the pen down. The `stroke` command actually draws the line segments traced out by the turtle's pen. The `showpage` ejects the page from the printer.



Question 3

There are many ways to solve this problem, iterative and recursive. Here are 3 possible solutions.

```
link find0(link x)
{ while(x != NULL && x->key != 0) x = x->next;
  return x;
}
```

[1]Copyright 1999, Kevin Wayne.

```
link find0(link x)
{ for(; x != NULL && x->key != 0; x = x->next)
    ;
  return x;
}

link find0(link x)
{ if (x == NULL || x->key == 0) return x;
  return find0(x->next);
}
```

Question 4

This can be done with recursion or a loop. The recursive version is similar to Question 4, Midterm 2, Fall 1998 – the only difference between is that we only want to delete the first 0, not all of them. So if we find a 0 key, we just return the rest of the list, instead of the 0-free version of the rest of the list. (In practice, we should also worry about freeing the memory associated with the deleted node.)

```
link delete0(link x) {
   if (x == NULL) return NULL;
   if (x->key == 0) return x->next;
   x->next = delete0(x->next);
   return x;
}

link delete0(link head) {
  link x;
  if (head == NULL) return NULL;
  if (head->key == 0) return head->next;
  for (x = head; x->next != NULL; x = x->next)
    if (x->next->key == 0) {
      x->next = x->next->next;
      break;
    }
  return head;
}
```

Question 5

144

The data structure `test` is a node-like structure: each "node" has a key and two pointers to other nodes. The first few lines of code define the 3 variables `x, y` and `z` to be pointers to nodes. Each of their keys is initialized to 1. Also, each of the 3 variables has two pointers, one pointing to each of the other two variables.

To figure out what gets printed, it's convenient to create a table and keep track of each variable at every step. Only variables `t`, `x->k`, `y->k`, and `z->k` change, so we record these values at the beginning of each iteration through the `for` loop. It is not hard to recognize the pattern - Fibonacci numbers. Figuring out exactly which Fibonacci gets printed out requires a little care.

t	x	y	z
x	1	1	1
y	2	1	1
z	2	3	1
x	2	3	5
y	8	3	5
z	8	13	5
x	8	13	21
y	34	13	21
z	34	55	21
x	34	55	89
y	144	55	89
z	144	233	89

At the end of the last iteration of the `for` loop, `z->k` gets incremented to 377, then `t` is reset to `x`. So now `t->k >= 100` and the `for` loop terminates. At this point `t->key = x->key = 144` so 144 is printed. Note that the program actually computes all of the Fibonacci numbers less than 100, and then the next 3 Fibonacci numbers. But it prints out only the smallest Fibonacci number bigger than 100.

Question 6

Try out a few values and you will quickly see the pattern:

$$\text{strange}(x) = \begin{cases} 0 & x \leq 0 \\ x - 1 & x \geq 1 \end{cases}$$

Here's a formal argument. Obviously if `x` is nonpositive, `strange(x) = 0`. If `x` is positive and odd, `strange(x) = x - 1`. So what happens if `x` is even and positive. Well, `strange(x) = 1 + strange(x-1)`. Since x is even, `x - 1` is odd; thus, so `strange(x-1) = x - 2`. Hence `strange(x) = x - 1` in this case as well.

Question 7

First let's write down the truth table for the function given.

x	y	z	f
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Using the standard sum-of-products method, the Boolean function can be expressed as `f = x'y'z' + x'yz + xy'z + xyz` and drawn accordingly using 4 AND gates, 5 NOT gates, and 1 OR gate. Note also that since `(x+y) = x'y + xy' + xy`, we could express `f = x'y'z' + (x+y)z`. Thus, we could actually draw the circuit using only 2 AND gates, 3 NOT, and 2 OR gates. The circuit can be improved further by observing that `x'y' = (x+y)'`. Cleverness pays.

Question 8

This should be pretty straightforward by now.

- preorder: `A D F - - - B C - E G - - - -`
- inorder: `- F - D - A - C - G - E - B -`
- postorder: `- - F - D - - - G - E C - B A`
- level-order: `A D B F - C - - - - E G - - -`

Question 9

4

This is essentially the same as Question 13, Midterm 2, Fall 1997. The only difference is the base case - a `NULL` tree has the value 1. The easiest way to answer this question is to hand-simulate the `check` function, starting at the bottom of the tree. The value of the `check` function for node `x` increases (by 1) from one level in the tree to the next only if neither child of `x` is `NULL`.

In general, here's what the function computes. We say that a node is *fertile* if it has two non-NULL children. (We adopt the convention that a `NULL` node is fertile - they just aren't mature enough to have any children.) The function `check(x)` returns the maximum number of fertile nodes on any simple path from `x`.

Question 10

insertion sort, mergesort, quicksort

Quicksort requires roughly $N \log N$ time on a randomly ordered file of N keys (Sedgewick, Property 7.2). It requires roughly N^2 time on an already sorted or reverse sorted file (Property 7.1). We note that this N^2 time bound can be improved to $N \log N$ using a variant of quicksort that selects the partitioning element more carefully (uniformly at random or median).

Mergesort requires roughly $N \log N$ time on *any* input file, regardless of whether it is sorted, random, reverse sorted, etc. (Property 8.1). However, empirically, quicksort is typically twice as fast as mergesort on randomly ordered input files (Table 8.1, Table 9.2).

Insertion sort requires roughly N^2 time on a randomly ordered file (Property 6.2). It requires only roughly N time on a sorted file (Property 6.4). It requires N^2 time on a reverse sorted file.

Question 11

Another pretty familiar question by now.

Question 12

`E H I O C O I E R R U S S V T S`

By convention, we chose to partition on the last element - `R`. Also for duplicates, we adopt the convention that both pointers stop. Empirically, this leads to a better balancing of the partitions. So, in this example, we chose to exchange `R` and `E` in the third step.

```
T H I S C O U R S E I S O V E R      T-E
E H I S C O U R S E I S O V T R      S-O
E H I O C O U R S E I S S V T R      U-I
E H I O C O I R S E U S S V T R      R-E
E H I O C O I E S R U S S V T R      S-R
E H I O C O I E R R U S S V T S
```

Question 13

```
int sum(link x)
{
  if (x == NULL) return 0;
  return x->key + sum(x->l) + sum(x->r);
}
```

A classical divide-and-conquer recursive program. Clearly our function should return an integer, and the input should be a link to the first node in the tree. As usual with linked lists and trees, the base case is an empty tree. In this case, we return 0. The simple, but very critical, observation is that the sum of the keys in a tree rooted at `x` is precisely the sum of the following three things (i) `x->key`, (ii) the sum of the keys in the left subtree, and (iii) the sum of the keys in the right subtree.

Question 14

37

The best way to answer the question is to start computing the `sum` values starting at the bottom of the tree and working your way up the tree, just as in dynamic programming.

Here's a more complete explanation as to what goes wrong and why the function doesn't compute the sum of all the keys, as you might first expect. The structure is no longer a tree. Some nodes have two incoming arc; hence their `sum` gets double counted. Actually it is a bit more complicated. The number of times the key of a node gets counted is equal to the number of different paths from the root to that node. E.g., the bottom key 3 will get counted five times, since there are 5 different paths. We should point out that the recursive function from Problem 13 would be essentially useless on such an input (assuming we actually had a use for the funny looking sum anyway). Why? In general, there can be (exponentially) many paths, so our function would recompute the same thing over and over. A dynamic programming approach would be preferred.

Question 15

	0	1	
0	1	2	start
1	0	1	
2	1	2	accept

The key is associating a meaning with each state. State 0 represents bit strings with an even number of 0's that end in a 0. State 1 represents bit strings with an odd number of 0's. State 2 represents bit strings with an even number of 0's that end in a 1. Given this interpretation, the FSA is straightforward to construct.

It's good practice (but not necessary here) to determine the corresponding RE. Here it is (1*01*01*)*1*1. The term in parentheses matches all bit strings with exactly 2 zeros. We can replicate this any number of times to get an even number of zeros. We need the final 1 to make sure the bit string ends with 1. The 1* is needed to match strings with no zeros, like 111111.

Question 16

First, figure out what languages the nondeterministic FSA accepts. Write down some sample strings in the language: `{1, 11, 111, 1111, 00, 100, 1001, 1001111, 11101111011, ...}`. Note that 000 is not accepted because there is no transition possible from state 2 if the next bit is a 0. FSA's must end up exactly in the accept state(s) and use up all of the input characters. The nondeterministic FSA accepts all bit strings composed of all 1's or bit strings with exactly 2 0's, i.e., those corresponding to the regular expression `11* + 1*01*01*`.

From lecture, we know that it is always possible to design a deterministic FSA that expresses the same language as a nondeterministic FSA. We construct such a deterministic FSA. Here it is convenient (and necessary) to allow multiple accept states.

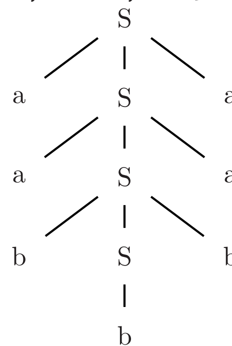
	0	1	
0	2	1	start
1	2	1	accept
2	3	2	
3	4	3	accept
4	4	4	

State 1 represents bit strings with all 1's. State 2 represents bit strings with exactly one 0. State 3 represents bit strings with exactly two 0's. State 4 represents bit strings with three or more 0's. State 0 is really only needed so that we don't accept the empty string; otherwise we could delete state 0 and make state 1 the starting state.

Question 17

There are 4 grammar rules: (i) $S \to a$, and (ii) $S \to b$, (iii) $S \to aSa$, and (iv) $S \to bSb$. The grammar is context-free (Type II), but not regular (Type III). We start by listing some of the strings in the language, and see that all odd length palindromes are accepted.

`{a, b, aaa, aba, bab, bbb, aaaaa, aabaa, ababa, abbba, ...}`



Question 18

regular

Recall from Lecture 15.5 that we created an FSA to accept bit strings that, if interpreted as binary numbers, are divisible by 3. Using the same idea, we could test divisibility by 99 (or any other number). Also, recall that FSA's are equivalent to RE's. Hence any language accepted by an FSA is regular (Type III). Of course, we could also express the given language using a Type 0 grammar, but that would be akin to describing a 1941 Mouton Rothschild as just a bottle of red wine! The Chomsky hierarchy classifies different types of grammars in order of their ability to describe more languages. Each type can express more languages than the previous.

Type	Grammar	Corresponding Machine
III	regular	FSA
II	context-free	nondeterministic PDA
I	context-sensitive	linear bounded automata
0	recursive	Turing machines

Question 19

B

The decision version of the traveling salesman problem (*Does there exists a tour of length at most L?*) is NP-complete. This means that it is "computationally equivalent" to every other NP-complete problem. I.e., if you had an algorithm that could guarantee to solve *any* TSP problem "efficiently," then you could solve any NP-complete problem "efficiently." Of course if we have a large problem (say with 10,000 nodes) we expect the algorithm will take much longer. By efficiently, we mean that the number of elementary steps does not grow too quickly as a function of the input size, e.g., $N^2, N \log N$. An algorithm is said to be *efficient* or *polynomial-time* if its running time grows no more rapidly than some polynomial function of the input size. Most computer scientists conjecture that NP-complete problems are intractable: no efficient algorithm is likely to exists for NP-complete problems, including the TSP.

This does not mean we can not hope to solve particular instances of the TSP problem. The conjecture really only says that there exist some hard instances that we won't be able to solve efficiently. But maybe these hard instances are pathological, and we have no interest in solving them anyway. In fact, there are many real-world TSP problem with some additional structure that enable us to solve them efficiently. Unfortunately, there are also some real-world TSP problems (with less than 1000 nodes) that no one has been able to solve!

- A - Suppose the n points are all on a single line. Then finding an optimal tour is trivial. NP-complete problems can have many easy instances.
- B - This is the conjecture that most computer scientists believe for all NP-complete problems.
- C - This TSP problem is not one of the unsolvable problems (like the Halting problem). The recursive algorithm given in Lecture 9.6 will find an optimal tour. It essentially enumerates all $(n-1)!$ possible tours. Of course, this algorithm takes way too long to be of any practical use.
- D - Using cleverness, it is possible to eliminate many tours without explicitly considering them. Suppose the n cheapest links in the network form a tour; then this is a provably optimal tour, and there is no need to look further.
- E - This statement is essentially equivalent to A. Nothing prevents an algorithm from "getting lucky" and being able to finish early for some input instances.

Question 20

0014

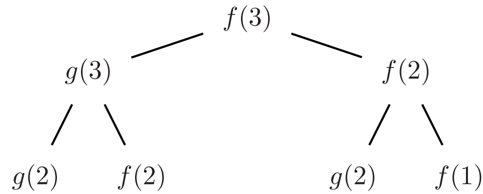
The first four lines initialize registers R1 = 0, R2 = 1, R3 = 4, and R4 = 5. Line 14 increments R1 by 1 each time it is executed, since R2 is never modified. The instruction 7414 and 7313 (jump and count) are then used to perform a double nested loop. Instruction 7414 creates an "inner loop" which increments R1 five times. Instruction 7313 creates an "outer loop" which repeats the inner loop four times. Thus, R1 is incremented 5*4=20 times in total, so the final value of R1 is the hex number 0014.

Question 21

TOY instructions only allocate 2 hex digits (8 bits) for indexed addressing. To access $2^{11} = 2048$ words (instead of $2^8 = 256$), we would need 3 more bits. Thus, instructions would have to be longer than 4 hex digits (or we would have to make do with fewer instructions).

Question 22

$f(3) = 9$. It takes 20 calls to g to compute $f(3)$. Here's the start of the tree. You can figure out the rest.



x	$f(x)$	$g(x)$
0	1	1
1	2	1
2	4	2
3	9	5
4	22	13
5	56	34
6	145	89

The above table actually lists more values than you need to solve the problem. I computed `f(20)` using this program. After about a minute and over 500 million mutually recursive calls to `f` and `g` it finally returned the answer. Only 41 different values of `f` and `g` are computed, so almost of all the work is redundant and unnecessary. A dynamic programming approach would solve this problem much more efficiently. Actually, these types of mutually recursive functions (also called recurrence relations) can be evaluated analytically, but this requires some advanced math or Maple. It turns out that,

$$f(x) = 1 - \frac{1}{\sqrt{5}}\left(\frac{2}{3+\sqrt{5}}\right)^x + \frac{1}{\sqrt{5}}\left(\frac{2}{3-\sqrt{5}}\right)^x.$$

and that the number of calls to `g` required to compute `f(x)` is

$$\frac{5-3\sqrt{5}}{10}\left(\frac{2}{3+\sqrt{5}}\right)^x + \frac{5+3\sqrt{5}}{10}\left(\frac{2}{3-\sqrt{5}}\right)^x - 1$$

Question 23

other methods in this class

Question 24

```
int i, max = 0, count[51];
for (i = 0; i <= 50; i++) count[i] = 0;

for (i = 0; i < N;  i++) count[a[i]]++;

/* now find the value(s) that occur most often */
for (i = 0; i <= 50; i++)
  if (count[i] > max) max = count[i];
for (i = 0; i <= 50; i++)
  if (count[i] == max) printf(``%d\n'', i);
```

For efficiency, it is essential to access each element in the array `a[N]` only once, since `N` is assumed to be huge. We maintain an auxiliary array `count[51]` to count the number of occurrences of each key in the range `0` to `50`. We actually walk through this auxiliary array 3 times, but this cost is negligible relative to walking through the huge array `a[N]`.