COS 126 **General Computer Science** **Spring 1999**

Final - Solutions[1]

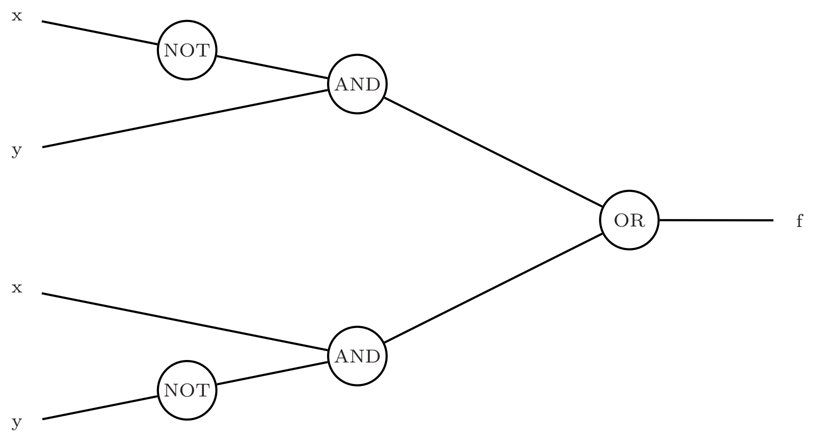
1. Convert the decimal numbers 77 and 23 to binary; take their bitwise XOR; then convert the result to octal. Circle your final answer. (Recall the XOR function is 1 if its two input bits are different, and 0 if they are the same.)

   132

   $77_{10} = 1001101_2, \;\; 23_{10} = 0010111_2$
   1001101 XOR 0010111 = 1011010.
   $1011010_2 = 001\ 011\ 010_2 = 132_8$.

2. Draw a Boolean circuit that computes the XOR function using only AND, OR, and NOT gates.

   Using the sum-of-products method, the Boolean formula is $f = x'y + xy'$. The resulting circuit is:



3. This problem considers whether it will be possible to solve large problems on future computers using an exponential algorithm (e.g., for the traveling salesman problem). Exponential functions grow extremely quickly, so this seems unlikely. However, computers are increasing in speed at an exponential rate too, so maybe this is enough.

   For concreteness, assume that computers double in speed every year (18 months is more realistic). Also, assume that today's computer can perform 2^{100} steps in an hour (a very generous estimate).

   (a) Suppose you have a 2^N algorithm for the problem. How big of a problem could you solve on today's computer in 1 hour? What about on a computer built in 10 years?

   100, 110

   If a computer performs x instructions per hour, then we want to solve for N in the equation $2^N = x$. With today's computer $x = 2^{100}$; in 10 years $x = 2^{100} \times 2^{10} = 2^{110}$.

[1]Copyright 1999, COS 126.

(b) How many years will pass before a computer is fast enough to solve a problem of size $N = 1000$ in 1 hour using a 2^N algorithm?

900 years

We need $x = 2^{1000}$. If computer double in speed every two years, then in k years $x = 2^{100} \times 2^k = 2^{100+k}$. This equals 2^{1000} when $k = 900$.

(c) Redo part (a) assuming you have an N^5 algorithm. (Use $2^{10} \approx 1,000$ to simplify your numbers.)

$2^{20} \approx 1$ million, $2^{22} \approx 4$ million

Solve for N in the equation $N^5 = x$.

4. Label each of the following statements `true`, `false`, `likely`, or `unlikely`. Assume the Church-Turing thesis is true, and that P $\neq$ NP is likely. TSP is NP-complete; PRIME is in NP. Unless otherwise specified, assume polynomial and exponential refer to the number of steps on a deterministic Turing machine.

`true`	There exists an *exponential* algorithm for TSP.
`unlikely`	There exists a *polynomial* algorithm for TSP.
`true`	There exists a *polynomial* algorithm for TSP on a *nondeterministic* Turing machine.
`false`	There exists an *exponential* algorithm for the halting problem on the *R2D2 computer*.
`true`	Discovering a polynomial algorithm for TSP would imply the immediate discovery of a polynomial algorithm for PRIME, but maybe not vice versa.

5. For each of the 10 terms on the left, choose the best matching description on the right.

b	compiler	(a)	prepare program for compiler
i	linker	(b)	translate program from high-level language to machine language
j	loader	(c)	translates from assembly language to machine language
c	assembler	(d)	high-level language simulation to execute program without translation
d	interpreter	(e)	traverses parse tree in postorder to generate code
a	preprocessor	(f)	represents structure of computation
g	lexical analyzer	(g)	converts input into stylized stream of tokens using finite state automata
h	syntax analyzer	(h)	builds parse tree from sequence of tokens using pushdown automata
f	parse tree	(i)	concatenates input object modules to make a single output file, merges external references, and merges relocation tables
e	code generation	(j)	translates object code to executable code by adding start address to each address requiring relocation

6. Give the TOY machine code that might be generated by a compiler for the C statement:

```
a = (b + c) * c;
```

Assume a, b, and c are 16-bit integers stored at locations AA, BB and CC, respectively. Start your code at location 20, and use at most 5 TOY instructions.

```
20: 92BB   R2 <- mem[BB] = b
21: 93CC   R3 <- mem[CC] = c
22: 1123   R1 <- R2 + R3 = b + c
23: 3113   R1 <- R1 * R3 = (b + c) * c
24: A1AA   mem[AA] <- R1
```

7. Consider the following TOY program. Assume that the following numbers are loaded into memory (all other location are set to 0000), and that the machine is started with pc = 10.

```
10: B0A0  R0 <- A0             D0: 0001  D8: 0006  E0: 0003
11: B1D0  R1 <- D0             D1: 0003  D9: 0006  E1: 0005
12: 5018  goto 18              D2: 0002  DA: 0006  E2: 0005
13: 9B05    R3 <- mem[R0 + R5] D3: 0005  DB: 0006  E3: 0004
14: B701    R7 <- 1            D4: 0004  DC: 0006  E4: 0000
15: 1227    R2++               D5: 0005  DD: 0006  E5: 0000
16: 1337    R3++               D6: 0005  DE: 0006  E6: 0000
17: AB05    mem[R0 + R5] <- R3 D7: 0004  DF: 0006  E7: 0000
18: 9D12    R5 <- mem[R1 + R2]
19: 6513    R5--; if (R5 > 0) goto 13
1A: 0000  halt
```

What values does it leave in memory locations A0, A1, ..., A7?

memory location	A0	A1	A2	A3	A4	A5	A6	A7
value	0000	0001	0001	0002	0003	0005	0008	0000

Think of the integers stored in memory starting at D0 as data. The program reads in these integers in order (using indexed addressing mode in instruction 18: 9D12) until an entry with value 0000 is reached (instruction 19: 6513 checks for this condition). Memory locations A1 - A7 is an array that counts the number of times each integer (between 1 and 7) appears in the data. E.g., when data element D6: 0005 is processed, A5 is incremented by one. This program is similar to the histogram program from Midterm 1, Spring 1999.

8. Suppose that we connect a 3-bit counter to a multiplexer, and the output of the multiplexer to the input of a second 3-bit counter as follows:

Assume that both counters are initially set to all zeros. Give the values of a_2, a_1, and a_0 after 8 clock ticks.

tick	a_2	a_1	a_0
0	0	0	0
1	0	0	1
2	0	1	0
3	0	1	0
4	0	1	1
5	1	0	0
6	1	0	0
7	1	0	0
8	1	0	1

On the ith clock pulse, the multiplexer selects the ith input line. If it is 1, then the multiplexer outputs 1. This has the effect of incrementing the second counter by 1. If the multiplexer outputs 0, then the input to the second counter is 0, so nothing changes.

9. Consider the mutually recursive C functions.

```
int f(int x) {
  if (x == 0) return 1;
  return f(x-1) + g(x-1);
}

int g(int x) {
  if (x == 0) return 2;
  return g(x-1) + f(x-1);
}
```

(a) What does `f(g(2))` evaluate to?

96

Here's a table of `f(N)` and `g(N)` values:

	0	1	2	3	4	5	6
f	1	3	6	12	24	48	96
g	2	3	6	12	24	48	96

In general, for $x \geq 1$, $f(x) = g(x) = 3 \times 2^{x-1}$. We conclude $f(g(2)) = f(6) = 96$.

(b) How many mutually recursive function calls are required to compute `f(N)`? Circle the best answer.

i. $O(\log N)$
ii. $O(N)$
iii. $O(N \log N)$
iv. $O(N^2)$
v. $\mathbf{O(2^N)}$

10. Consider all binary search trees containing the following 15 keys:

```
U S E T H E F O R C E L U K E
```

(a) Suppose the 15 keys are inserted into an empty BST in the given order. When inserting a duplicate key, always put it in the right subtree of any matching keys. Draw the BST. What is its height? (Recall, the height of the empty tree is 0, and height of a tree with one node is 1.)

8

(b) Give the inorder traversal of the BST in (a).

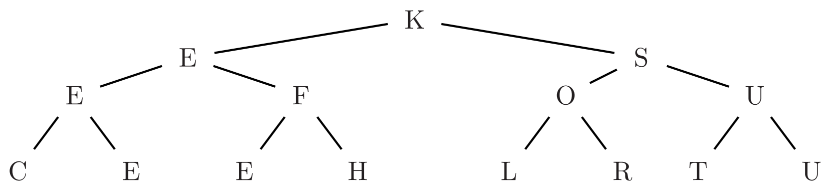
`C E E E E F H K L O R S T U U`

The inorder traversal of a BST always sorts the keys.

(c) What is the minimum height among all BST's containing these 15 keys? Draw such a BST.

4

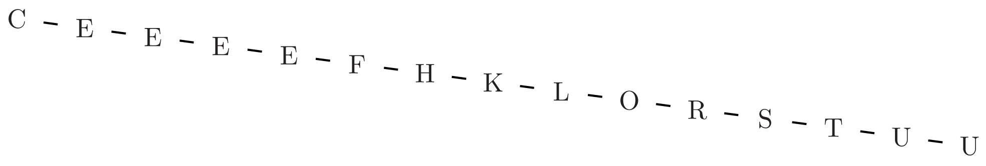
Choose the median element as the root, and do this recursively in each subtree. E.g., the initial root node will be `K`. For any input with N keys, this procedure results in a BST with height $\lceil \log_2(N+1) \rceil$. You can't do any better.



(d) What is the maximum height among all BST's containing these 15 keys? Draw such a BST.

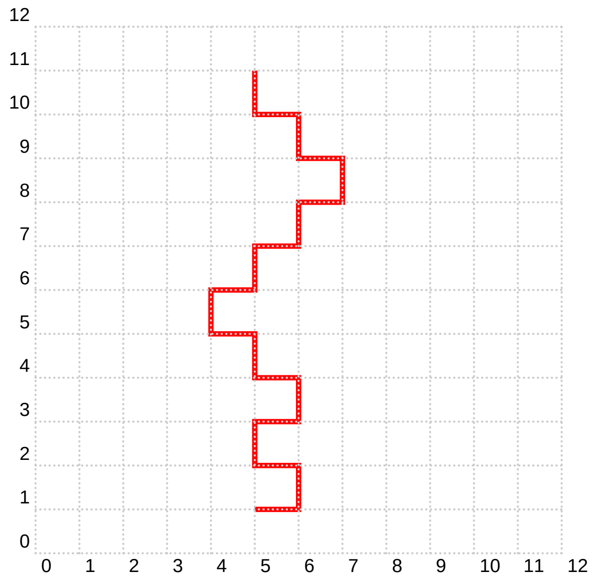
15

Sort the keys in increasing order. Insert the keys into an empty BST in this order. It will look just like a linked-list.



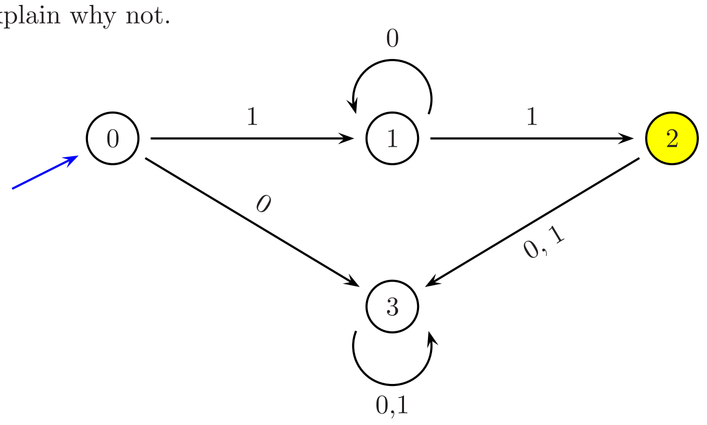
11. What does the following PostScript program draw? Use the grid below to record your answer. Recall, '`N` {...} `repeat`' repeats the grouped sequence `N` times.

```
%!
5 1 moveto
0 0 1 1 1 0 0 1 0 1
10 { 2 mul -1 add 0 rlineto 0 1 rlineto} repeat
stroke
showpage
```



12. Consider the language generated by the regular expression `1(0*)1`.

(a) Does there exists a deterministic FSA that recognizes the language? If yes, draw such an FSA. If no, explain why not.

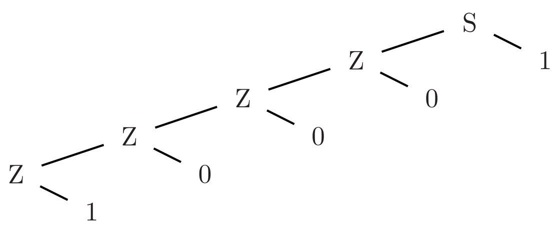


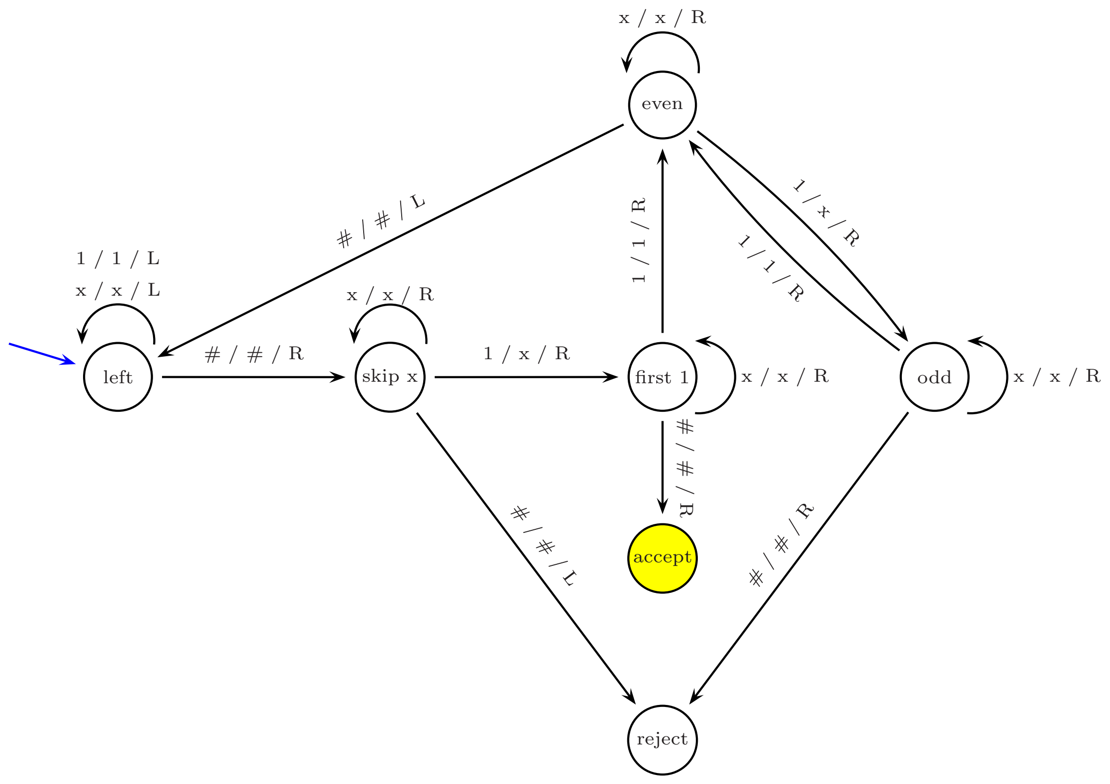
(b) Does there exists a Type III (regular) grammar that generates the same language? If yes, give such a grammar. If not, explain why not, and give a Type II grammar.

terminals	$0, 1$
nonterminals	S, Z
start	S

Rules
$S \to Z1$
$Z \to Z0$
$Z \to 1$

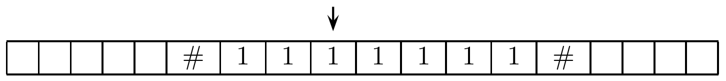
(c) Give the parse tree for the string `10001` using the grammar from part (b).



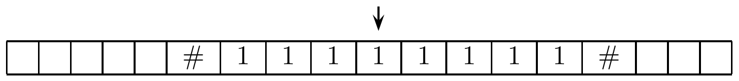


13. Consider the Turing machine above. The alphabet consists of the three characters: `1`, `x`, and `#`. For parts (a) and (b), suppose the Turing machine is started with the input tape and initial cursor as given; circle the input if it will be accepted.

(a)



(b)



(b)

(c) Characterize which inputs (i.e., characters between the two delimiting #'s) are accepted by the Turing machine.

All tape inputs whose number of `1`'s is a power of 2 (e.g., 1, 2, 4, 8, 16) are accepted, Why is this the case? Each pass from left to right through the tape skips all characters other than `1`. It "deletes" every other `1`. By delete, we mean that it replace the character `1` with the character `x`. If the number of `1`'s is odd (and bigger than 1), it rejects; otherwise we go back to the leftmost `#` and repeated the process. If the original problem has N `1`'s, then after one pass through the tape, the new problem has $N/2$ `1`'s. If N is a power of 2, then so is $N/2$.

Here's a description of each state.

- `accept, reject`: self-explanatory
- `left`: Move the cursor one position to the left, until it points to the leftmost `#`. If it currently points to the leftmost `#`, then transition to `skip x` and move the cursor one position to the right, i.e., to the beginning of the interesting portion of the tape.
- `skip x`: Move the cursor to the right, until it reaches the first character that is not `x`. If the first such character is `#`, transition to `reject`; if it is `1`, the transition to `first 1`.

- `first 1`: Skip over any `x`'s while moving the cursor to the right. If the first character scanned (other than `x`) is `#` then transition to `accept`, as there has been exactly one `1` scanned during this pass through the tape. If it is `1`, then transition to state `even`.
- `even`: An even number of `1`'s has been scanned during this pass through the tape. Skip over any `x`'s. If the cursor points to a `1`, delete it and transition to `odd`; otherwise go transition back to `left` and start the whole process over.
- `odd`: An odd number of `1`'s has been scanned during this pass through the tape. Skip over any `x`'s. If the cursor points to a `1` transition to `even`; if it is `#`, transition to `reject`.

(d) How many steps (in the worst-case) does it take the Turing machine to accept or reject an input with N consecutive 1's delimited by two #'s? Circle the best answer.

i. $O(\log N)$
ii. $O(N)$
iii. $\mathbf{O}(\mathbf{N}\log\mathbf{N})$
iv. $O(2^N)$

Traversing the tape from left to right (or right to left) takes $N+2$ steps, since there are N 1's and two #'s. Half of the remaining `1`'s are deleted each time you go from left to right through the tape. Thus, you do this at most $\lceil \log_2 N \rceil$ times. You go through the tape from right to left once for each left to right pass, so this doubles the number of steps. The total number of steps is $O(N \log N)$.

14. Suppose that a binary tree is made up of nodes of type

```
typedef struct node* link;
struct node {int key; link l; link r; };
```

Write a recursive C function `int height(link x)` that takes as input a link to the root of the binary tree, and returns its height.

```
int max(int a, int b) {
  if (a > b) return a;
  else return b;
}

int height(link x) {
  if (x == NULL) return 0;
  return 1 + max(height(x->l), height(x->r));
}
```

15. Suppose that a linked list is made up of nodes of type

```
typedef struct node* link;
struct node {int key; link next; };
```

Consider function `swap(x, y)` that allegedly swaps the nodes pointed to by `x->next` and `y->next`. Assume that neither `x->next` nor `y->next` is the last node of the list, and that neither `x` nor `y` is the first node on the list.

```
void swap(link x, link y) {
  link xn, xnn, yn, ynn;

  xn = x->next; xnn = x->next->next;
  yn = y->next; ynn = y->next->next;

  x->next = yn; yn->next = xnn;
  y->next = xn; xn->next = ynn;
}
```

Explain why the code does not work as claimed. Be brief but specific. **Hint: draw a picture.**

The code breaks down if the two nodes are next to each other in the linked list, i.e., `x = y->next` (or `y = x->next`). (Otherwise, it works fine, even if `x = y`.) To see this, note that `xn = ynn`. The last statement in `swap(x, y)` is `xn->next = ynn`. This creates a self-loop, a node pointing to itself.

16. Given an array `int a[N]`, with k consecutive positive integers followed by $N - k$ consecutive zeros, your goal is to find the value k.

    E.g., if `a[ ]` $= \{5, 6, 12, 33, 18, 17, 3, 0, 0, 0, 0, 0, 0, ..., 0\}$, then $k = 7$.

    (a) Write a C function `findk1(int a[ ])` that returns the value k in $O(k)$ steps.

    ```
    int findk1(int a[ ]) {
       int k;
       for (k = 0; k < N; k++)
         if (a[k] == 0) return k;
    }
    ```

    (b) Write a C function `find2(int a[ ], int l, int r)` so that `find2(a, 0, N-1)` returns k in $O(\log N)$ steps. Give a brief description of how your function works.

    ```
    int findk2(int a[ ], int l, int r) {
      int m = (l + r) / 2;
      if (r == l + 1) return r;
      if (a[m] == 0) return findk2(a, l, m);
      return findk2(a, m, r);
    }
    ```

    Use bisection search. The function maintains the invariant that `a[l] > 0` and `a[r] == 0`. Initially the interval `[l, r] = [0, N-1]`. Each iteration decreases the interval size in half, so there are $O(\log N)$ iterations.

    (c) Write a C function `findk3(int a[ ])` that returns k in $O(\log k)$ steps. Give a high-level description of how your function works.

    ```
    int findk3(int a[ ]) {
      int r;
      for (r = 1; r < N; r *= 2)
        if (a[r] == 0) return find2k(a, 0, r);
      return find2k(a, 0, N-1);
    }
    ```

    The key is to first obtain a value `r` that is within a factor of 2 of `k`. Then, we binary search on the interval `[0, r]` to find k. This interval is of size at most $2k$, so it takes $O(\log k)$ steps. We find the value `r` using *geometric doubling*. At each iteration we increase `r` by a factor of 2 as long as `a[r] != 0`. Thus, we find a value `r` such that `a[r/2] > 0` and `a[r] == 0`. Now, $r/2 < k \leq r$ as desired.

17. Write a Java `class Circle` for circles. Represent a circle by its center point and its radius. Use `class Point` from Lecture for Euclidean points. Implement method `boolean contains(Point p)` that returns `true` if the Point `p` is properly contained inside the circle, and `false` otherwise. Then, implement method `double dist(Point p)`: if `p` is contained in the circle return 0.0; otherwise return the Euclidean distance between `p` and the nearest point on the boundary of the circle. (Do not worry about writing a constructor.)

```
class Point {
  private double x, y;

  Point(double x, double y) {
    this.x = x; this.y = y;
  }

  double dist(Point p) {
    double dx = this.x - p.x;
    double dy = this.y - p.y;
    return Math.sqrt(dx*dx + dy*dy);
  }
}
```

```
class Circle {
  private Point center;
  private double radius;

  public boolean contains(Point p) {
    return (center.dist(p) < radius);
  }

  public double dist(Point p) {
    if (contains(p)) return 0.0;
    return center.dist(p) - radius;
  }
}
```