Algorithms

1. Let $X[1, \ldots, n]$ be an array of n positive integers, and let $O[1, \ldots, n-1]$ be an array of $n-1$ symbols drawn from the set $\{+, *\}$. We can form an arithmetic expression by interleaving the two arrays, like this:
$$X[1] \text{ O[1] } X[2] \text{ O[2] } \cdots \text{ O[n-2] } X[n-1] \text{ O[n-1] } X[n]$$
Notice, however, that we can get different values by inserting balanced parentheses into this expression at different places. For example, if $X = [1, 2, 3, 4, 5]$ and $O = [+, *, +, *]$, some of the choices are:
$$\begin{aligned} 1 + (2 * (3 + (4 * 5))) &= 47 \\ (((1 + 2) * 3) + 4) * 5 &= 65 \\ (1 + 2) * (3 + (4 * 5)) &= 69 \\ (1 + 2) * ((3 + 4) * 5) &= 105 \end{aligned}$$
Your objective is to determine how large the value of the expression can become. In the example above, the last parenthezation turns out to be optimal, so the correct answer is 105.

   Specifically, you should **describe** an algorithm that, given X and O, computes the largest value that can be achieved by inserting balanced parentheses into the expression. Your algorithm must run in $O(n^3)$ time. (You may assume that arithmetic operations on the input numbers take constant time.) **Explain** why your algorithm works, in enough detail to convince an intelligent but skeptical reader that it is correct. Your algorithm only needs to output the *value* obtained by the optimal parenthezation; it does *not* need to output the optimal parenthezation itself.

2. Three arrays A, B, and C of positive integers, each containing n elements sorted into increasing order, are given. Assume that there is least one element common to all three arrays. That is, assume there exist indices i, j, and k, such that $A[i] = B[j] = C[k]$.
   - **Describe** an algorithm that finds this common element in $\Theta(n)$ time. If there is more than one common element, your algorithm may return any of them. **Explain** why your algorithm works, in enough detail to convince an intelligent but skeptical reader that it is correct.
   - **Describe** how to extend your algorithm to the more general problem in which there are m input arrays, instead of exactly three. What is the run time of your extended algorithm?

3. **Find** tight asymptotic bounds on any positive real-valued function $T(n)$ satisfying the following recurrence for all sufficiently large n:
$$T(n) = T\left(\frac{n}{3}\right) + T\left(\frac{n}{6}\right) + T\left(\frac{n}{9}\right) + n$$
That is, find an expression $f(n)$, as simple as possible, such that $T(n) = \Theta(f(n))$. Use the substitution method to **prove** that your answer is correct. (Note: Implicit floors or ceilings in the recurrence do not affect the answer.)

Spring 2018 CSE Qualifying Exam
CSCE 531, Compilers

1. **LR-Parsing.** Consider the following augmented grammar G with start symbol S':

$$\begin{array}{lcl} S' & \rightarrow & R \\ R & \rightarrow & L \ = \ R \\ R & \rightarrow & L \\ L & \rightarrow & * \, R \\ L & \rightarrow & a \end{array}$$

   (The idea is that R stands for "r-value" and L stands for "l-value.")

   (a) For the grammar G above generate all of the LR(1) sets of items $I_0, I_1, I_2, \ldots, I_{10}$ (eleven states in all) along with complete transition information for a canonical LR(1) parser.

   (b) Using the sets-of-items constructed in part (b), construct the action table and describe any conflicts. Assume the productions are numbered in order from 0 to 4.

   (c) Describe in detail how an arbitrary LR parsing algorithm will proceed in general when the next token is t and the stack contents are $s_0, s_1, \ldots, s_{top-1}, s_{top}$.

2. **Syntax-Directed Translation.** The do-alternatingly statement with grammar below:

$$S \ \rightarrow \ \textbf{doalternate } S_1 \textbf{ or } S_2 \textbf{ until } B$$

   executes S_1 then tests B, if this is false then S_2 executes followed by a test of B again. As long as B evaluates to false this statement alternates between S_1 and S_2; when B evaluates to true the loop is exited. Give semantic actions necessary for generation of intermediate or assembly code for this statement.

3. **Control Flow and Liveness Analysis.** The following fragment of 3-address code was produced by a nonoptimizing compiler:

```
 1 start:  sum = 0
 2         i = 1
 3 test:   if i > n goto end1
 4 switch: goto disp
 5 case1:  sum = sum + 1
 6         goto end2
 7 case2:  t1 = i
 8         t2 = 2 * t1
 9         t3 = t2 - n
10         sum = sum + t2
11 case3:  t1 = sum
12         t2 = t1 + 2
13         sum = t2
14         goto end2
15 default:sum = sum + 1
16         goto end2
17         goto end2
18 disp:   if i == 5 goto case1
19         if i == 8 goto case2
20         if i == 13 goto case3
21         goto default
22 end2:   i = i + 1
23         goto test
24 end1:   print sum
```

Assume that there are no entry points into the code from outside other than at `start`.

(a) (20% credit) Decompose the code into basic blocks $B_1, B_2, \ldots$, giving a range of line numbers for each.

(b) (30% credit) Draw the control flow graph, describe any unreachable code, and coalesce any nodes if possible.

(c) (30% credit) Give a table with 24 rows saying which variables are live immediately before each line number. Assume that `n` and `sum` are the only live variables immediately after line 24.

(d) (20% credit) Describe any simplifying transformations that can be performed on the code (i.e., transformations that preserve the semantics but reduce (i) the complexity of an instruction, (ii) the number of instructions, (iii) the number of branches, or (iv) the number of variables).