2) (Sum 14) Dynamic Programming (Matrix Chain Multiplication)

Using the dynamic programming algorithm shown in class, determine the minimum number of multiplications to calculate the matrix product ABCD, where the dimensions of A, B, C and D are given below:

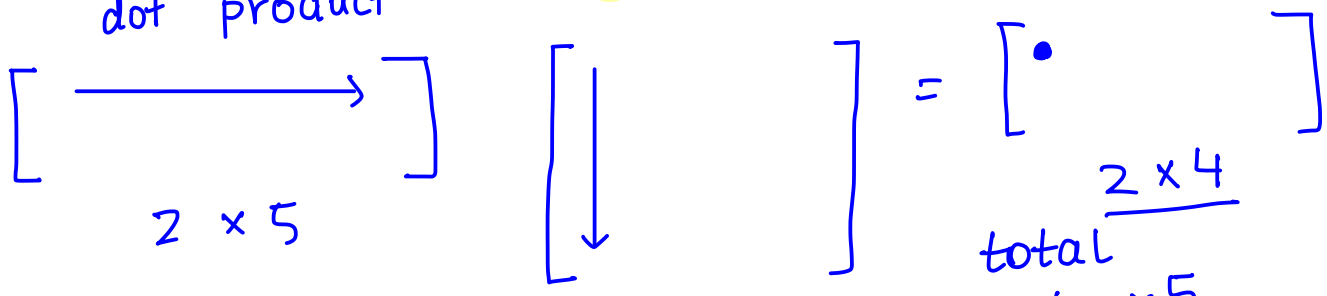
A: 2 x 5, B: 5 x 4, C: 4 x 1, D: 1 x 5

	A	B	C	D
A	0	40	30	40
B	X	0	20	45
C	X	X	0	20
D	X	X	X	0

Notes:

Matrix multiplication: A (2 x 5) B (5 x 4)

dot product



operations: 2 x 4 x 5 (total each)

AB = 2 x 5 x 4 = 40

BC = 5 x 4 x 1 = 20

DC = 4 x 1 x 5 = 20

ABC

(AB)C = 40 + 0 + 2 x 4 x 1 = 48

A(BC) = 0 + 20 + 2 x 5 x 1 = 30

BCD

(BC)D = 20 + 0 + 5 x 1 x 5 = 45

B(CD) = 0 + 20 + 5 x 4 x 5 = 120

ABCD

(ABC)D = 30 + 0 + 2 x 1 x 5 = 40

(AB)(CD) = 40 + 20 + 2 x 4 x 5 = 100

A(BCD) = 0 + 45 + 2 x 5 x 5 = 95

4) (5 pts) What is the fewest number of comparisons necessary to sort 11 numbers via a comparison sort? Note: 11! = 39,916,800 and $2^{23} = 8{,}388{,}608$.

size n, $n!$ permutations

to find sorted order (correct position) in all permutations $\log_2(n!)$ Binary search

total comparisons $k = \log_2(n!)$

$2^k = n! \Rightarrow 2^k \geq 11!$

$2^2 = 4$

$2^2 (8e6 < 2^{23} < 9e6)$

$2 (32e6 < 2^{25} < 36e6) < 11!$

$(64e6 < 2^{26} < 72e6) > 11!$

$k = 26$

2) (6 pts) Consider running Karatsuba's algorithm for multiplying two eight-bit integers, X and Y, shown below, by breaking both down into two four-bit parts. The algorithm entails computing three four-bit multiplications, one of which could have a tiny bit of overflow. For the following example, show the three recursive multiplications that would be calculated using Karatsuba's:

```
X = 01011101
Y = 10110110
```

$X_h = 0101$ $X_\ell = 1101$

$Y_h = 1011$ $Y_\ell = 0110$

In each of the slots below, write **in binary** the two numbers that would have to be multiplied in the three recursive calls. Product #1 is the one that may end up having 5-bit number(s) in it. (Hint: The only actual computation that needs to be done in this problem is a tiny bit of binary addition.)

X_h 0101
$+ X_\ell$ 1101
10010

Y_h 1011
Y_ℓ 0110
10001

Product #1: 10010 x 10001 $(X_h + X_\ell) \times (Y_h + Y_\ell)$

Product #2: 0101 x 1011 $X_h \times Y_h$ high bits

Product #3: 1101 x 0110 $X_\ell \times Y_\ell$ low bits

5) (9 pts) Show the result of running the dynamic programming algorithm to find the edit distance between the strings "ACGATTACGA" and "CTAGACTTGA" in the table below. The first row has been filled in for you.

	C	T	A	G	A	C	T	T	G	A
A	1	2	2	3	4	5	6	7	8	9
C	1	2	3	3	4	4	5	6	7	8
G	2	2	3	3	4	5	5	6	6	7
A	3	3	2	3	3	4	5	6	7	6
T	4									
T	5									
A	6									
C	7									
G	8									
A	9									

edit distance (strings s, t):
turn s into t

① if char match, diagonal (no use) & no operation

② else operation change

$1 + \min$ { leave s last char (left); leave t last char (up); leave both last char (diag.) }

7) (Sum 14) Dynamic Programming – Zero/One Knapsack Problem

Find the maximum valued knapsack of size 12 or less choosing from the following items (only 1 copy of each item is available). To get credit, please use the algorithm shown in class. Here are the items from which you are choosing:

Item	Weight	Value
Apple	3	4
Banana	4	6
Cantaloupe	7	13
Durian	5	9
Emblic	2	5
Fig	1	3

Show the algorithm by filling in the following table:

base case [0] = 0

total wt.

Item	1	2	3	4	5	6	7	8	9	10	11	12
Apple	0	0	4	4								
Banana	0	0	4	6								
Cantaloupe	0	0	4	6								
Durian	0	0	4	6								
Emblic	0	5	5	6								
Fig	3	5	8	8								

The correct answer to the query should be in the bottom right square of the table.

MAX

take item [item-1] [totalWt − itemWt] + itemValue

- [item-1]: current item not taken
- [totalWt − itemWt]: satisfy this item wt based prev. state
- itemValue: so add this value

OR

leave ⇒ above row (current item not included) same column ↳ same total wt.

6) (Sum 14) Dynamic Programming – Longest Increasing Sequence

Assume that you've already written a method LCS, that calculates the length of the longest common subsequence between two sequences of integers. (Its prototype is given below.) Write a method that takes in 1 sequence of integers and calculates its longest increasing sequence. For ease of implementation, assume that the input only contains distinct integers. Note: You may use both the Arrays.copyOf and Arrays.sort methods. These are listed below. (Note: The code is pretty short.)

```
// Returns an array storing the first newLength elements of original.
int[] copyOf(int[] original, int newLength);

// Sorts the specified array into ascending numeric order.
void sort(int[] a);


public static int lis(int[] seq) {
        // other array increasing
        // common subsequence is increasing
        int[] sorted = Arrays.copyOf(seq);
        Arrays.sort(sorted);
        return lcs(seq, sorted);



}

public static int lcs(int[] x, int[] y) {

    int[][] table = new int[x.length+1][y.length+1];

    for (int i = 1; i<=x.length; i++) {
        for (int j = 1; j<=y.length; j++) {
            if (x[i-1] == y[j-1])
                table[i][j] = 1+table[i-1][j-1];
            else
                table[i][j] = Math.max(table[i][j-1], table[i-1][j]);
        }
    }
    return table[x.length][y.length];
}
```

6) (6 pts) The array below is the completed path array from running Floyd Warshall's Algorithm on a graph with vertices labeled 0 through 6. Using this array, write down the requested shortest paths. To write down a shortest path from vertex a to vertex b, write a sequence of 2 or more vertices, starting with a, ending with b, where each pair of consecutive vertices represents an edge on the shortest path between a and b. (Note: the array headers (indexes) are in bold and the contents of the array are not.)

From/To	**0**	**1**	**2**	**3**	**4**	**5**	**6**
0	0	2	3	4	0	2	5
1	6	1	3	1	0	2	5
2	6	2	2	1	0	2	5
3	6	2	3	3	0	2	5
4	6	2	3	4	4	2	5
5	6	2	3	6	0	5	5
6	6	2	3	6	0	2	6

Note: You may not use all the slots provided.

Shortest Path from 0 → 6: 0 , 4 , 3 , 2 , 5 , 6 , ____

Shortest Path from 5 → 4: 5 , 6 , 0 , 4 , ____ , ____ , ____

Shortest Path from 1 → 0: 1 , 3 , 2 , 5 , 6 , 0 , ____

stored path array

→ vertex stored is coming from

11) (6 pts) Using the Master Theorem, write down the solution to the three recurrence relations shown below. (You should write down T(n) = O(?), where ? is a some function of n.)

(a) $T(n) = 4T\left(\frac{n}{2}\right) + O(n)$ $\log_2 4 = 2$, $n^1 <= n^2$ $O(n^{\log_b a}) = O(n^2)$

(b) $T(n) = 3T\left(\frac{n}{2}\right) + O(n^2)$ $\log_2 3$, $n^2 >= n^{\log_2 3}$ $O(f(n)) = O(n^2)$

(c) $T(n) = 64T\left(\frac{n}{4}\right) + O(n^3)$ $\log_4 64 = 3$, $n^3 = n^3$ $O(n^{\log_b a} * \log n) = O(n^3 \lg n)$

Master Theorem: (divide-n-conquer)

$$T(n) = a\,T\left(\frac{n}{b}\right) + f(n)$$

- a: # subproblems
- b: each subproblem size
- $f(n)$: outside recursion

asymptotic bounds of $T(n)$:

<= ① if $f(n) = O(n^{\log_b a - \epsilon})$, $T(n) = \theta(n^{\log_b a})$

= ② if $f(n) = \theta(n^{\log_b a})$, $T(n) = \theta(n^{\log_b a} * \log n)$

>= ③ if $f(n) = \Omega(n^{\log_b a + \epsilon})$, $T(n) = \theta(f(n))$

$\epsilon > 0$, constant

You are given n k-sided fair dice, each labeled 1, 2, 3, …, k. You roll all of them. Then, separate out the ones that show k. Take the rest and roll them all again. Then, separate out the ones of these that show k, and roll the rest again. Repeat this process until you've separated all the dice out (ie, they all show k). The question we want to analyze is the number of times we expect to roll before completing the game.

9) (Spr 18) One way to analyze this is to write a simulation, run it many times and take the average. The latter portion of this is fairly trivial, so for this question, you will simply write a single Java method that takes in n, the number of dice, and k, the number of sides on each dice, simulates this process, and returns the number of turns it took to complete a single simulation of the game. Fill in the method signature given below. Assume that you have access to a static class variable r, that is of type Random, which you can use to generate random integers.

```
public static Random r;

public static int numTurnsSim(int n, int k) {
        // count turns until all n dice == k
        // based on random roll

        int turns = 0;
        while (n > 0) { // left dice
                int numk = 0; // this turn count
                for (int i=0; i<n; i++) {
                        int curRoll = r.nextInt(k)+1;
                        if (curRoll == k) numk++;
                }
                turns++;
                n -= numk;
        }
        return turns;
}
```

3) (7 pts) Consider counting the number of ways to make change for 14 cents using 1 cent, 3 cent, 5 cent and 8 cent coins. Fill in the table below showing the work that the dynamic programming algorithm to solve the problem would do to solve the problem. The first row has been filled out for you for convenience.

amt.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
3	1	1	2	2	2	3	3	3						
5	1	1	2	2	3	4	4	5						
8	1	1	2	2	3	4	4	6						

base case [curCoin][0] = 1 no coins

take coin if curCoin <= amt, add based prev.

[curCoin-1][amt] + [curCoin] [amt-curCoin]

OR

exclude curCoin & make amt

OR

include curCoin to make remaining change

leave curCoin > amt

[curCoin-1] [amt]

(prev state cur coin cannot be used)

fewest coins

take min([curCoin-1][amt], [curCoin][amt - curCoin])

leave [curCoin-1] [amt]

base case

dp[0][0] = 0

dp[0][j] = ∞

dp[i][0] = 0 zero coins → amt 0

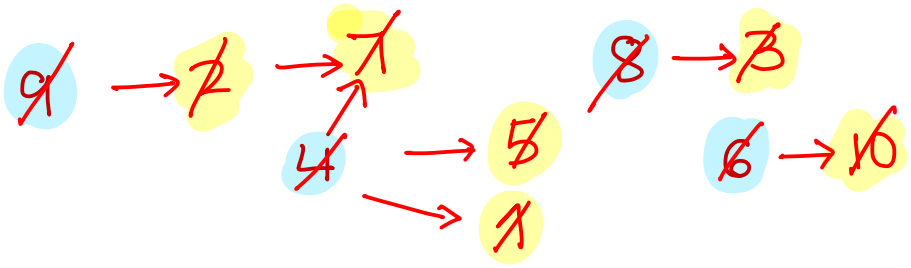
11) (Sum 14) Graphs – Topological Sort

Alice has to complete items 1 through 10. The following ordered pairs show the dependency between the items she must complete. Namely, for each ordered pair (a, b) shown below, she must complete item a before item b.

(2, 7), (8, 3), (9, 2), (4, 5), (4, 1), (4, 7) and (6, 10).

Show the ordering of the items produced by the algorithm shown in class that builds the list from the front and always adds "safe" nodes iteratively. When choosing between multiple possible "safe" nodes, always add the lowest numbered one first.

4 , 1 , 5 , 6 , 8 , 3 , 9 , 2 , 7 , 10



unlock before go

12) (6 pts) A common tactic to solve some type of problems is a technique called "Coordinate Compression". In this technique, we start with a set of input being a set of unique integers that aren't consecutive. (For example, the set could be {5, 2, 9, 16, 3, 12}.) For each number in the set, we would like to know its 0-based rank. When we sort the set, the order of the values is 2, 3, 5, 9, 12 and 16. The information we would like to be able to retrieve easily is the rank of any number in the list. For example, the rank of 2 is 0 and the rank of 12 is 4. To be able to retrieve this information, we would like to create a HashMap, mapping each number to its rank. In this example, the map would store 2 → 0, 3 → 1, 5 → 2, 9 → 3, 12 → 4 and 16 → 5. Write a static method that takes in an array of long values (assumed to be distinct) and returns a HashMap<Long,Integer> storing this mapping.

```
public static HashMap<Long,Integer> compress(long[] vals) {
    // sort, loop & give rank based on i
    Arrays.sort(vals);
    HashMap<Long, Integer> map =
            new HashMap<Long,Integer>();
    for (int i=0; i < vals.length; i++) {
        map.put(vals[i], i);
    }
    return map;
}
```

12) (Sum 14) Algorithm Design – Dynamic Programming

Describe a dynamic programming algorithm (in words) to solve the following problem:

Soccer players are ordered 0, 1, 2, …, n – 1, where n is a positive integers less than or equal to 100. Each player i ($0 \le i \le n-2$) can pass to any player j such that $j > i$. The probability the pass will succeed is prob[i][j]. (Assume that this information is given in the input.) As with all probabilities, note that $0 \le prob[i][j] \le 1$, for all $0 \le i < j \le n - 1$.

We are allowed to choose any sequences of passes in order to move the ball from player 0 to player n – 1. (This sequence will necessarily be some subsequence of 0, 1, 2, …, n – 1, since all the players can only pass the ball in "one direction.") If all players act optimally, design an algorithm that calculates the probability that player n – 1 successfully receives the ball.

sub problem:

- f(i) = maximal probability of player i receiving ball
- calculate in increasing values of i

⇒ 1D array memoization

base case $f(0) = 1$, ball starts

- for some f(i), know previous f(x) where $0 \le x < i$ (some player x before)
- so calculate probability for all x & take maximum

$$f(i) = \max(f(x)\, prob[x][i]),\ 0 \le x < i$$

calculate for each i from 1 to n-1

Run time: $O(n^2)$