CS490 Quiz 3

NAME: ___________________________

STUDENT NO: _____________________

SIGNATURE: __________________________

This is the written part of Quiz 2. The quiz is closed book; in particular, no notes, calculators or cell phones are allowed.

Not all questions are of the same difficulty, so be sure to pick up the easy points first, before tackling hard problems. The quiz is divided into two parts:

Part 1: [15]

Answers in part 1 should be succinct. Erroneous “extra” comments may decrease your score.

Part 2: [30]

When designing algorithms in part 2, identify the important algorithmic and implementative concepts. The use of pseudocode is greatly encouraged! There is no need to write working code. Feel free to quote algorithms we described in class, but make sure you describe how the algorithm is used. (e.g. to quote Dijkstra, you should specify your source node and where the results are stored, say an array. Then, go on to explain what you do with those results.) You can also quote the complexities of the algorithms discussed in class

NOTE: feel VERY free to use the backside of the exam papers.
NOTE: the bonus questions at the back are more difficult.

Part 1 - Short Answers [5 points each]:

1. Describe a situation or problem in which memoization HAS to be used instead of iterative DP. Also describe a situation or program in which iterative DP HAS to be used instead of memoization. (p.s. hard to code IS NOT a good reason)

[2.5] Iterative is needed if:

   a. Memoization causes stack overflow (recursion is too deep), OR
   b. State space is too big, and memoization uses too much memory, while there may be a smart way of filling in the table such that we don't need to keep the whole table in memory (recall our backpack example)

[2.5] Memoization is needed if:

Table to big for iteration, but is sparse in practice. Memoization will fill in only the few table entries needed, and some sparse memory storage (e.g. a map) would solve the memory problem.

2. Give a POSITIVE (> 0) integer solution pair to 39x + 65y = 104.

65 = 39*1 + 26
39 = 26*1 + 13
25 = 13*2 + 0

So, 13 = 39-26 (second line) = 39 - (65-39) (first line) = 2*39 – 65
So multiply by 8 gives 104 = 16*39 – 8*65. (x = 16, y = -8)

To make it positive, perturb the x by 5 (= 65/13), y by 3 (=39/13)
[5] This gives the only positive integer solution pair (16 - 3*5, -8 + 3*3) = (1, 1)

3. In class we described the backpacker problem. Given a set of N objects with weights <= W, value <= V, we need to pick the most valuable subset that has total weight less than B, our backpack capacity. We then discussed an O(N*B) DP solution. Sometimes, however, we may have V << B. Give an O(N*V) solution. CORRECTION: V = maximum value in total.

Have an array of size V. The i^{th} element keeps track of the lightest load with value i. Then, we update the array N times, each time with an object in our set.

```
int weights[N], values[N];  // assume data is already stored there
int lightestLoad[V+1];      // assume filled with infinity
lightestLoad[0] = 0;
for(int object = 0; object < N; object++)
    for(int value = V; value >= 0; value--)
        if (value >= values[object])
            lightestLoad[value] <?=
              lightestLoad[value-values[object]]+ weights[object];

Now loop through the value array one more time for the maximum value
that can be realized with weight less than W.
```

Part 2 [10 points each]:

1. Shopping.

Life is very convenient in year 5000, since you can get everything from the leading (and only) convenience store - MicroStore. You need to buy a list of (distinct) gifts, a_1, a_2 … a_n, for Xmas (n <= 15).

To save money, you can buy some of the gifts in packages. MicroStore offers a range of gift packages for different prices (at most 1000). For example, the following may be on sell:

Package	Price
1, 2	$4
2, 3	$3
1, 3	$2

Then, to buy gifts 1, 2, and 3, you would only spend $5 dollars. Given the gifts you neeulate the cheapest way to do Xmas shopping. You may assume the following.

1. You can buy more than you need, as long as that makes it cheaper
2. Everything you need is sold in one or more packages
3. Packages only contain items you need (assume you will throw out other unwanted items)

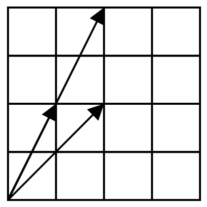
[8] DP on the items already bought or items still needed.

[2] Discuss bitmasks as an implementation technique.

e.g.

```
int packBitMasks[1000]; // bit j of element i is set if ith package sells item j.

int packCost[1000];     // package costs


int dp[1 << 15];        // bitmasks storing what is bought, filled with infinity
dp[(1 << 15)-1] = 0;

for(int mask = (1 << 15)-1; mask >=0; mask--)
    for(int pack = 0; pack < 1000; pack++)
        dp[mask] <?= dp[mask & packBitMasks[pack]] + packCost[pack];

dp[0] then stores the answer.
```

2. Distinct Rays
   Given N, consider the integer lattice points (x, y) on the Cartesian plan such that 0 <= x, y <= N. (i.e. we have an N+1 by N+1 lattice). We are interested in rays that originate from (0, 0) and point toward one of the lattice points above. Before we examine their mathematical beauty, however, we are interested in how many distinct rays there are.



Given N <= 40000, describe an algorithm that computes the number of distinct rays as described above. (e.g. When N = 4, the ray pointing at (1, 2) is the same as the ray point at (2, 4), but different from the ray pointing at (2, 2), illustrated on the side).

O(N^2) algorithm would be too slow, and earn only part marks.

[5] Recognize that rays coincide if the coordinates of the rays are multiples of each other. e.g. (2, 4) = 2*(1, 2), so they are the same way.
(If you now go through all N^2 pairs, reduced then to lowest form, and counted the number of unique elements, say, with a set, you get 5 points)

[2] Further recognize that you just need to count pairs that are relatively prime

[3] Realize the you just need to sum up the phi function to count the number of relatively prime pairs. Formula can be a little off and still receive full credit.

# of rays = $3 + 2 \cdot \sum_{i=2}^{N} \phi(i)$
(the 3 in the beginning counts (1, 1), (0, 1) and (1, 0)).

-OR-

[5] Use inclusion exclusion.
[2] Include all pairs.
Exclude all pairs where the gcd is divisible by a prime
Include all pairs where gcd is divisible by two distinct primes
etc…

[3] First prime factorize every number. For any number k, the number of pairs that has gcd divisible by k is (N/k+1) * (N/k+1) (since both number must be a multiple of k, counting 0). We discard any number whose prime factorization has any prime repeated. (e.g. 50 = 2*5*5 is discarded). Numbers that are the product of an odd number of distinct primes are included, while numbers that are the product of an even number of distinct primes are excluded.
(e.g. include 2, 3, 5, 7, 11, 13, …
      exclude 6, 10, 14, 15,
      include 30, 42, ….)

3. Split and Merge.
   We define an operation on an integer, N (<= 1000), code named “split-and-merge,” as follows:

   a) First, split N into many integers a_1, a_2, …, a_k such that

   $N = a_1 + a_2 + \ldots + a_k$

   b) Next, we merge them into a new integers M by multiplying them together

   i.e. $M = a_1 * a_2 * \ldots * a_k$

   Given N <= 1000, how many distinct M can we arrive at by split-and-merging N? (e.g. N = 5, we can get {1, 2, 3, 4, 5, 6}, a total of 6 distinct values. To get 6, do 2*3)

[3] Realize that we can count M by counting prime factorization. (Remember? We said unique prim factorization is important)
[2] Realize that we can use DP to count number of prime factorizations.
[2] DP on (number left, smallest prime allowed)
[3] Work out details (see below).

```
splitAndMerge(number, smallestPrime) {
    return 1 if smallestPrime > number (we just split the rest of number into 1’s).

    otherwise we can try to use smallestPrime multiple times
    answer = 0;
    for(int i = 0; i*smallestPrime < number; i++)
        answer += splitAndMerge(number – i*smallestPrime, nextPrime);
    return answer;
}
```

`where nextPrime is the next prime. e.g. if smallestPrime is 7, nextPrime is 11.`

BONUS [10] Project dog relief.
By year 6000, civilization has crumbled upon itself after MicroStore had a financial crisis. Thus, you have an army of dogs to defend your family from wild beasts. However, it is quite problematic when the dogs need to go (err.. eliminate solid waste). You have a lawn field discretized into a 1000 by 10 grid. Some grids are taken by trees and rocks, but the remaining free grids can each accommodate one dog that needs to go. There is one more constraint though; a dog won't relieve itself if any of its four neighboring squares are "in use" by other dogs. Given the map of the field (i.e. which grids are blocked), calculate the maximum number of dogs that can be relieved simultaneously.

R	D	
		D
D	R	

R		D
	D	
D	R	D

Sample lawn and dog placements (D = dog, R = Rocks)

[5] Realize that to place dogs on a row, we only need to know about the situation of the previous, current, and next row.

[3] DP row by row. i.e. DP on (row Number, bitmask of available spots of this row)

[2] work out recurrence, similar to below.

```
dp(rowNumber, okaySpots) { // okay spots is a bitmask storing which places can a dog go
    best = 0;
    foreach (place of dogs on each row that uses only okay spots) {
        if some pair of dogs are neighbors on this row, continue to next placement;

        calculate the okaySpots of next row. A square on the next row is okay
        if there is no tree, and that we didn't put a dog above it in the current
        row.

        best >?= dp(rowNumber+1, okaySpotsOfNextRow);
    }
    return best;
}

Of course, memoize. DP array is 1000 by (1<<10), ie. 1024000.
```

4. MORE BONUS [3]
   Suppose we encrypt by

   x -> x^e (mod n)

   Decrypt by

   y -> y^d (mod n)

   If the public key is (n, e) = (85, 7) and the secret key is (n, d) =(85, d), find d.

We need e * d = 7d = 1 (mod phi(85)) (from theorem in class)
First, 85 = 5 * 17;
So, phi(85) = 85 * (4/5)*(16 /17) = 64;

so need 7d = 1 (mod 64)

7d + 64 n = 1;

Run egcd:

64 = 7*9 + 1;

so, 7 * (-9) + 64 = 1;
so d = -9 = 55 (mod 64)