```
//Problem 1
//1.1)
int minimumCost(int *size, float *cost, int k, int n) {

	float best[n+1] = infinity //set all values to infinity
	best[0] = 0.0
	for (int i = 1; i < n+1; ++i)
	{
		for (int j = 0; j < k; ++j)
		{
			if ((i - size[j]) < 0) {
				continue;
			}

			best[i] = min(best[i],best[i-size[j]] + cost[j])
		}
	}
	return best[n]
}

//1.2)
//this function assumes there is a way to make n pierogies
int *sizes(int *OPT, int *size, float *cost, int k, int n) {
	int *usedList = malloc(sizeof(int) * (n+1)) //won't be this large but just in case
	int used = 0

	int i = n
	while (i > 0) {
		for (int j = 0; j < k; ++j)
		{
			if ((i - size[j]) >= 0) {
				if (OPT[i] >= (OPT[i-size[j]] + cost[j])) {
					usedList[used] = size[j]
					used++
					i = i - size[j]
					break; //breaks out of the for loop. no need to finish all k
checks.
				}
			}
		}
	}
	usedList = realloc(sizeof(int) * (used+1))
	return usedList
}
```

/*
For part 1: function minimumCost
O(n*k) run time:
This can be seen clearly with the for loop inside another for loop.
i goes from 1 to n and j goes from 0 to k-1.

O(n) space:
array best takes n+1 space.

For part 2: function sizes
O(nk) run time. O(n) space.
i in its worst case goes from n to 0.
But in any average case it will skip many of the numbers from n to 0 because of i = i - size[j].
j always goes from 0 to k. This leads to O(n*k) run time.
usedList takes worst case size of n+1 but in most cases it's much less.
*/

```
//Problem 2
int maxnetvalue(int *v, int n, int m) {
	int n //length of rod
	int m //cost per cut
	int v[n+1] //value per length v[1] to v[n] are inputted values
	int best[n+1] //will calculate best value for length 0 to length n

	best[0] = 0 //base case: best value for length 0 is 0
	for (int i = 1; i < n+1; ++i)
	{
		best[i] = v[i]
	}


	for (int length = 3; length < n+1; ++length)
	{
		int numPairs = (int) ((length-1)/2)+1
		for (int j = 1; j < numPairs; ++j)
		{
			best[length] = max(best[length],v[length-1-j]+v[j]-m)
			//should have figured out lower cases with optimal cuts so won't
			  need to evaluate more than one cut
		}
	}

	return best[n]
}
/*
O(n^2) run time. This can be seen clearly with the for loop inside another for loop.
length can max be size n and j can max reach size (n-1)/2. n * (n-1)/2 is O(n^2) time.
initialization takes O(n) time
*/
```

```
//Problem 3
float averageWeight(int **edge, int n) { //edge is an adjacency matrix
	int cost[end+1] //really only needs to go from start to end. But it's easier to index this
way.
	int numPaths[end+1]

	//base cases:
	numPaths[n-1] = 1
	cost[n-1] = 0

	//initialize everything else
	for (int i = 0; i < n - 1; ++i)
	{
		numPaths[i] = 0
		cost[i] = 0
	}

	for (int i = n - 1; i >= 0; --i)
	{
		for (int j = i + 1; j < n + 1; ++j)
		{
			if (edge[i][j] != 0) {
				numPaths[i] += numPaths[j]
				cost[i] += (cost[j] + 1*numPaths[j]) //all edges are unweighted and
equal to 1
			}
		}
	}

	if (numPaths[0] == 0) {
		return NaN
	}
	else {
		return cost[0]/numPaths[0]
	}

}
/*
O(n^2) run time. This can be seen clearly with the for loop inside another for loop.
i is size n going from n -1 to 0 and j can max reach size n when i = 0.
initialization takes O(n) time
*/
```