# Problem Review for 2017 UCF Local Programming Contest

## Problem 1: Electric Bill

This is the easiest problem in the set, exemplifying a typical week 3 or week 4 Introduction to C programming assignment. We need a loop to run the multiple cases (as we do for all of the problems) and for an individual problem instance, we just need to set up an if statement to handle our two billing cases (a) used 1000 kwh or less, (b) used more than 1000 kwh. There are probably a few different ways to handle the case work. Either way, when charging the extra rate for the kwh above 1000, in the calculation, it's important to subtract 1000 from the total kilowatt usage.

## Problem 2: Simplified Keyboard

The cases that are easiest to discern are (a) the case when the two strings are identical and (b) the case when the two strings are different due to being different lengths. If writing a function that takes in two strings to solve the problem, it's useful to immediately handle these two cases (using the appropriate string functions or methods) so that we can focus on the only interesting case - two strings of the same length that are not identical. Since we make the same comparison between letters for each corresponding letter of the string, let's just focus on checking to see if two letters are "adjacent" or not on the keyboard. Once we solve this subproblem, we just want to check if EVERY pair of corresponding letters are adjacent or not.

One way to solve this that might be the easiest computationally but require a little extra hard-coding, would be to hardcode the neighbors for all 26 letters. If you take the time to hardcode this list, then for a given letter, you can just check to see if the corresponding letter in the other string is in its neighbor list.

Another way to solve the problem would be to write a function that extracts the (x,y) coordinate of any given letter (or store these in a table) and use this as a tool. For example if we want to see whether letters 'n' and 'v' are adjacent or not, we could get the coordinates of 'n' (1,4) and 'v' (2, 3). Then we could check to see if the absolute value of the differences between the x-coordinates and y-coordinates is less than or equal to 1.

A third method would be simply to start at the x-y coordinate of 'n' and use DX/DY arrays (DX = {-1,-1,-1,0,0,0,1,1,1}, DY = {-1,0,1,-1,0,1,-1,0,1}), and for each of the coordinates of the form (x + DX[i], y + DY[i]) see if the letter 'v' is stored in any of them. Notice that in the array I included (0, 0) because we consider the letter itself adjacent to the letter.

Of these methods, the second one is probably the fastest to implement.

## Problem 3: Singin' in the Rain

After reading the problem, one key realization is that after a track plays, the CD ends up queued to the beginning of the next track. So, we use this number as the current track for our calculation for the jump. For example, if we play track 43 on a CD with 44 or more songs, the CD starts at track 44 before we forward to the next song. The next realization to be made is that since so many tracks are allowed on the CD (up to 1 billion), one can't count one by one to the next track. Instead, subtraction must be used to figure out the number of button presses in one direction, and

then we can subtract this number from the total number of tracks to get the number of button presses in the other direction since the button presses in both directions form a circle of n presses, when there are n tracks on the CD. For example, if we are on track 44 (after finishing playing track 43) and we are attempting to move to track 15 on a CD of 70 songs, we would move |44 - 15| = 29 button presses backwards and 70 - 29 = 41 button presses forward. Of these, we want the smaller choice, in this case, 29. Finally, the numbers are big enough that the final result must be stored in a long in Java (or long long in C/C++).

**Problem 4: Editor Navigation**

This problem tempts readers into a greedy solution. While one may exist, many standard attempts can be shown to be incorrect via counter-example. It's fairly difficult to discover the optimal path for the first sample case by hand in fact! Rather than try to fine tune a greedy approach, an easier solution for a contest situation is to run a breadth first search. Each letter position on each line would be a potential location (vertex in an underlying graph) for which we are searching for a minimum distance to. From each location, we can press four keys, which will take us to four potentially new unvisited locations. It follows that the run-time of a straight-forward implementation would be O(n), where n represents the sum of the number of characters on each line, since the branching factor from any location is constant(no more than 4). Given the limits on the variable, a breadth first search easily runs in time. The most difficult part of the implementation (if one has coded a breadth first search before), is simply a function that correctly figures out which location you move to from a given location for each of the four button presses. Also, it's important to note that there's no need to actually form any graph structure. In a standard BFS, we must keep an array that stores the distance from the starting poing to each possible location. One can either use a one dimensional array where we map each (r,c) pair to a unique integer, or a two dimensional array of integers where each individual array is the size of each of the lines of text given in the input.

**Problem 5: Simple Darts**

For each dart, we just want to convert the (x, y) coordinate given into a distance from the center of the board and a reference angle (measured in the usual way). The distance from the center tells us whether we would score 50, twice the corresponding base score, or the corresponding base score. The reference angle can easily be mapped to the base score just based on what proportion around the circle it is (from the positive x-axis). The distance formula ($\sqrt{x^2 + y^2}$) will tell us how far from the center of the board a dart is and the atan2 function can be used to calculate this reference angle.

**Problem 6: Multimodal Transportation**

Anyone with an algorithms background (say COP 3503 at UCF) will probably recognize that a shortest distance algorithm ought to be used. It's tempting to create a graph with n vertices, where n is the number of cities in the input, but an issue occurs in this representation when dealing with changing the mode of transport. In reality, what we see is that within a city, we have four locations: an airport, a seaport, a rail station and truck station. Between these locations we have a travel cost (given in the input). Thus, our graph should have 4n vertices. For each city, we put in all possible edges within the different stations of the city all with the same associated cost of moving between any two stations within a city. Then, between cities, each input edge specifies a single edge in our graph between one pair of stations in the two cities that are being connected.

Since n = 400 is the maximum case, our graph could have up to 1600 vertices. A graph algorithm may remember to find shortest distances is Floyd-Warshall's algorithm. But this algorithm has a run-time of $O(V^3)$, where V is the number of vertices in the graph. For a programming contest $1600^3$ = 4.096 billion operations is too many to run in a reasonable run time. Since we must simply solve the shipping problem for a single source location, Dijkstra's algorithm (either $O(V^2)$ when written with a straight-forward implementation or O(ElgV), when using a priority queue) can be used to solve the problem fast enough.

**Problem 7: Videogame Probability**

Since the probabilities of success for each trial of a particular task are fixed and independent, there is no difference in the end result regardless of the order in which we attempt the tasks. Given this key realization, it's easiest to pretend to simply go through the tasks in order. Since the input data has frequencies of each task, we can "unroll" the input data into a larger list of separate tasks with associated probabilities of success. The key subproblem to solve is given n attempts and k successes, what is the probability of attaining precisely k tasks after n total attempts at tasks. The is dependent on the probabilities of solving k tasks in n-1 total attempts (followed by a failure on task k+1) and the probability of solving k-1 tasks in n-1 total attempts (followed by a success on task k). Recursively, we have the following formula (Let $p_k$ represent the probability of success on a single attempt at task k):

$$f(n,k) = f(n-1,k) \times (1-p_{k+1}) + f(n-1,k-1) \times p_k$$

If we code this problem recursively, it will get a time limit exceeded judgement due to the exponential nature of the branching factor. To speed up the code, one should use either memoization or dynamic programming (DP) where all results are stored in a two dimensional array. Once we store the answer to a recursive sub-problem, we should never make that recursive call again. Instead, we should immediately return the solution to that subproblem, saving time. In the DP solution, we build up our answers to subproblems from smallest to largest, avoiding recursion all together. Here is the crux of the recursive code:

```
for (int i=1; i<dp.length; i++) {
    for (int j=0; j<dp[i].length; j++) {
        double missTerm = j < tot ? dp[i-1][j]*(1-prob.get(j)) : dp[i-1][j];
        double makeTerm = j > 0 ? dp[i-1][j-1]*prob.get(j-1) : 0;
        dp[i][j] += (missTerm+makeTerm);
    }
}
```

**Problem 8: Maximum Non-Overlapping Increasing Subsequences**

A natural thought when looking at this problem is to solve the regular Longest Increasing Subsequence (LIS) problem for every contiguous subsequence of the input and see if we can build what the question is asking for with consecutive maximum increasing subsequences taken from a portion of the original input sequence. When we compute a single LIS for a sequence $s_0$, $s_1$, …, $s_{n-1}$, we naturally calculate the LIS for all sequences that start at $s_0$ and end at $s_i$, for all i, $0 \leq i \leq n-1$. This can be done in $O(n^2)$ time using a straight-forward dynamic programming algorithm. Since there are n possible starting points, we can compute the LIS for all possible contiguous subsequences in $O(n^3)$ time. Since n = 100, this pre-computation easily runs in time.

Now, consider trying to solve the problem assuming that all the values described above are pre-computed. For any given i and k, we would like to know the maximum number of terms we can include for the sequence $s_0$, $s_1$, …, $s_i$ given that each separate LIS must have at least k items in it. There are at most 100 possible values for both i and k, so we can store all possible answers to subproblems in an array of size no more than 10000. (Some of these slots never get used in a meaningful way because some cases are impossible, one can't have LIS's of length 10 or greater within the first 6 elements of the sequence, for example.)

Here is how to view the problem recursively: if we end at index i, our second to last subsequence of k or greater items must have ended at index j, where $j < i$, and our last LIS of length k or greater is within the sequence $s_{j+1}$, $s_{j+2}$, …, $s_i$. In this situation, we want the maximum number of terms we can grab up to index j added to the LIS on $s_{j+1}$, $s_{j+2}$, …, $s_i$, which is already pre-computed. Unfortunately, there are up to 100 possible break points for j (really i of them), so we simply must try all of these possible break points and take our best answer.

Here is my recursive function with memoization. Note that lis[i][j] stores the longest of all the longest increasing subsequences of the sequence $s_i$, …, $s_j$:

```
public static int solve(int n, int k) {

    if (lis[0][n] < k) return 0;
    if (memo[n][k] != -1) return memo[n][k];

    int res = lis[0][n];

    for (int i=1; i<=n; i++) {
        if (lis[i][n] < k) break;
        res = Math.max(res, lis[i][n]+solve(i-1,k));
    }

    memo[n][k] = res;
    return res;
}
```

**Problem 9: Rotating Cards**

A regular brute force solution would simply keep track of where each number in the list is (via a single for loop through the data and storing the results in an array), and then run for loops forwards and backwards going from number of number, adding all the terms in between to see which way is better. You always want to go the shorter way because no matter which way you go, the state of the cards is identical, so going the longer way once can never produce new savings later. So, in this way, the solution has a greedy structure. But, just to decide which way to go to get to the next card, we have to do up to $n = 10^5$ operations. We then have to repeat this $10^5$ times. Thus, the overall run-time of a brute force solution would be $O(n^2)$, and for $n = 10^5$ that means close to $10^{10}$ operations, which is too much for a contest.

Thus, a smarter solution is necessary. A Fenwick Tree (or Binary Index Tree), allows us to add or subtract a value from a single index in an array and query the sum of any consecutive elements in an array in O(lg n) time. In our problem, we essentially want to sum all the values in a range to determine which way to get to the next card. In the brute force solution, it takes us O(n) time to do this, but with a Fenwick tree, it would just take us O(lg n) time. To initialize the Fenwick tree with each card value takes is O(nlgn) time, because the run time to add a single item to an element of the array takes O(lg n) time and we have to repeat this n times. When using the Fenwick tree, we know the sum of ALL the elements in the card deck at any time. Thus, if we calculate the sum for rotating the cards forwards, we can deduce the sum of rotating the cards backwards by subtracting from the whole. (This is nearly identical to the logic in the Singin' in the Rain question.)

So, our solution is as follows: create a Fenwick tree of size n and fill it with input. As you do this, store a reverse look up table so you know which indexes each item, 1, 2, etc, are in. Loop through the reverse look up table, keeping track of the sum of the items left in the card deck. Initially, this is just $\frac{n(n+1)}{2}$, the sum of the first n positive integers. Then, use the reverse look up table to tell you from which index in the card list to which index in the card list you are moving. Put these two indexes into sorted order and make the query to the Fenwick tree of the sum of the items in that contiguous subsequence. Calculate the cost of going the other way by subtracting from the sum of the remaining cards. Add the smaller of these to your result. Then do some bookkeeping - subtract the sum of the cards left, subtract the card you discarded from the Fenwick tree, and update your current location. When you are done processing the cards, you'll have the optimal result!

**Problem 10: Multiples**

We first must make the observation that all we care about is the multiple of primes and product of primes. If a number is a multiple of 30, we would have already flagged it as a multiple of 2. So, we can ignore any composite number up to 130, when considering the list of values for which we want to look at their multiples. There are 31 prime numbers less than 130.

Our normal method of solution would have us add in all the values in the range divisible by 2, 3, 5, etc. But doing so overcounts values divisible by 6, for example, since we counted these one time when counting multiples of 2 and counted them a second time when counting multiples of 3. The generalized solution to solving this over-counting issue is the Inclusion-Exclusion Principle. The principle states that we want to add out items in the sets divisible by 2, 3, 5, etc. but then subtract out all the items divisible by a product of two distinct primes, then add in all the items divisible by a product of three distinct primes, then subtract out all the items divisible by a product of four distinct primes, and so forth. The standard implementation of this algorithm takes $O(2^n)$ time where n represents the number of distinct primes. In this problem we have up to n = 31 distinct primes and a run time of $2^{31}$ is too much for a programming contest.

What we quickly realize is that many of the products described above of some number of distinct primes easily exceeds our bound for b = $10^{15}$. In fact, the product of the first 14 smallest primes exceeds $10^{15}$. Thus, many of the $2^{31}$ subsets we would theoretically have to consider don't change our answer at all, since not a single value in the given input range would be divisible by those products.

Thus, the trick is to pre-compute ONLY the products of distinct primes < 130 that are ALSO less than $10^{15}$ in an efficient manner.

Furthermore, for different sub-problems, we may have various bounds for a, so it would be nice if we could sort our products of primes in such a way that we don't accidentally use a product of primes that is smaller than our b but contains a prime factor bigger than our a, for a particular query.

Finally, one last optimization is to note that all even numbers are divisible by 2 and it's very easy for us to get rid of these values and only solve the question for the odd values in range, adding in all the even values at the end. This optimization reduces the number of primes we consider as part of our general inclusion-exclusion to 30 from 31, which essentially divides by 2 the number of product of primes we will be generating in our pre-computation.

So, we will create 30 lists, one for each odd prime from 3 to 129, inclusive. The list for prime p will just store ALL the product of odd primes for which p is the LARGEST odd prime. The list for 3 just stores 3, since no other odd primes are less than 3. The list for 5 will store 5 and 15, all possible products for odd primes 5 or less that contain a 5. The list for 7 will store 7, 21, 35 and 105. Initially, these lists sizes will be doubling since we are taking all of the old lists and multiplying each value in all the old lists by the new prime and also including the new prime by itself. But, eventually, what will happen is that we'll start producing products that exceed $10^{15}$. When this happens, we just won't add these to our list and the list sizes will stop doubling.

Thus, we need a mechanism to store each of these lists and generate the subsequent list. To do this, we will keep a "master list" which has all of the values for the previous lists in sorted order. So, after processing 3, 5 and 7, we will store 4 lists in memory: the lists for 3, 5 and 7, as well as the master(M) list. Here are those lists, draw out visually:

3: 3
5: 5 → 15
7: 7 → 21 → 35 → 105
M: 3 → 5 → 7 → 15 → 21 → 35 → 105

Now, in our pre-computation, we want to build our list for 11, which we will do by simply starting it with 11, and adding to it each item in the master list multiplied by 11:

11: 11 → 33 → 55 → 77 → 165 → 231 → 385 → 1155

Now, we must update the master list. To do this, run the merge algorithm on our list for 11 and the old master list to obtain:

M: 3 → 5 → 7 → 11 → 15 → 21 → 33 → 35 → 55 → 77 → 105 →165 → 231 → 385 → 1155

It turns out that when you do this pre-computation, the total number of products we generate is 23,641,442. While this is a great number of products, we can typically do up to $10^8$ simple operations for a contest type problem, and this value is only about one quarter of $10^8$. One thing to notice is that we never had to run a sort explicitly. Since we maintain the sorted list property through multiple merges, each of which run in linear time, our total run-time for this precomputation is linear with respect to the total number of items created.

Once we do this precomputation, then for each a and b, we can just loop through all the prime lists corresponding to primes less than or equal to a. Then, for each product of primes, if it has an odd number of primes, we add (b/product + 1)/2, using integer division to our total, and if we have an even number of primes, we subtract the same value from our total. We divide by 2 to account for the fact that the range of values we are considering are odd numbers since we are dealing with all evens separately. The +1 is to work in the corner case. Say our product is 15 and b is 15, 1/2 is 0, but we want 2/2 in this case. Similarly, say b = 45 and our product is 15, we want to count both 15 and 45. (30 was counted with the evens.) In this case 45/15 = 3 and 3/2 is 1, but we want 4/2 to indicate that there are two odd values in 1..45 that are divisible by 15. Here is my code after the pre-comp, where div[i][j] represents, the $j^{th}$ product of primes for the list of the $i^{th}$ prime

```
long res = n/2; // All even values added in first.
for (int i=0; i<numP; i++) {  // Go through all prime lists.
    for (int j=0; j<div[i].length; j++) {
        if (div[i][j] > n) break;  // Stop in a single p list.
        if (parity[i][j])     res += (n/div[i][j]+1)/2;
        else                  res -= (n/div[i][j]+1)/2;
    }
}
```

**Problem 11: k-Item Shopping Spree**

Let's first consider a brute force solution to the problem. Let's say we have 4 different items in the shopping spree and we can choose 3 of them, in any order. There are 4 x 4 x 4 = 64 possible shopping sprees. We could use recursion or a triple for loop for this specific example to generate the value of every shopping spree (each value would just be adding the values of the three items selected) and then determine how many of the 64 shopping sprees added up to the query value.

It should be fairly obvious for the bounds given, that implementing this sort of brute force solution would simply not run in time, since we may have up to 10000 different items for the spree and we may select up to 10000, for a total number of $10000^{10000}$ number of shopping sprees, which is really $10^{40000}$ (1 followed by 40,000 zeroes).

A few weird things to notice about the input specs: the query values are limited to $500.00 dollars, meaning that we are limiting queries to 50,000 cents even though many sprees may well exceed that value, the mod value is extremely low, and the value of each item is fairly strictly limited as well (to no more than $100.00). It seems strange to limit the value of items and queries so strictly if our solution involves just storing values in variables. All of these strange issues in the bounds indirectly indicate that instead of storing the value of each item in a variable, that we may need to store how many items there are of each value in an array with the index of the value. Thus, if our input has three five cent items, instead of storing 5 three times, we will want to store the number 3 in the array index 5, and the array index implicitly represents the value of those 3 items. Now, consider that this array is nothing but a polynomial. Let's say that we have 2 items worth 1 cent, 1 item worth 4 cents, and 3 items worth 5 cents. Our array looks like:

| Index | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| Value | 0 | 2 | 0 | 0 | 1 | 3 |

Now, think of this as representing the polynomial $2x + x^4 + 3x^5$.

Now, consider squaring this polynomial:

$$(2x + x^4 + 3x^5)(2x + x^4 + 3x^5) = \begin{array}{lll} 4x^2 & + 2x^5 & + 6x^6 \\ 2x^5 & + x^8 & + 3x^9 \\ 6x^6 & + 3x^9 & + 9x^{10} \end{array}$$

If we carefully think about the meaning of each of the 9 terms in this polynomial, we see that $4x^2$ represents four shopping sprees of two items with value 2 cents - grabbing either of the 2 one cent items for the first item, followed by grabbing either of the 2 one cent items for the second item. The next term $2x^5$ on the first row represents grabbing either of the 2 one cent items for the first item, followed by the four cent item for the second item. Thus, the 2 in front of the $x^5$ here represents the two ways we can have a shopping spree of 2 items that add up to 5 cents where the first item is 1 cent and the second item is 4 cents. To pick one last example, consider the first term on the bottom row, $6x^6$, this represents the six shopping sprees of value six cents where the first item is one of the three items of value 5 cents and the second item is one of the two items of value one cent.

Thus, what we can do to solve the problem is take the input and store it as a polynomial of degree 10,000. (Technically, we'll store it with a higher degree, but we'll get to that in a bit.) If we were to just square this polynomial, the coefficient to $x^i$ would represent the number of shopping sprees of two items worth i cents. Generalizing this reasoning, what we'd like to do then is take our polynomial, P(x), as previously described, and raise it to the power k, the number of items to be selected in the shopping spree, and then for each term with an exponent 50,000 or less, we need to extract the coefficient mod 997.

As we multiply a few times, since our queries go to 50,000, at a minimum, in our polynomial, we have to store it to degree 50,000. The regular algorithm for polynomial multiplication runs in $O(n^2)$ time. So if we were to use this algorithm, it would do roughly $50000^2 = 2.5$ billion operations, too much for a programming contest. To speed up polynomial multiplication, we can use the Fast Fourier Transform, which runs in O(nlgn) time. A limitation of the FFT is that since it uses doubles, it has limited accuracy - this explains why the mod value is so low.

So, any time we multiply two polynomials, where the coefficient to $x^i$ in the polynomial represents a number of shopping sprees of value i, mod 997, we will be using the FFT.

Our second problem is that an exponent as high as 10,000 seems to indicate calling the FFT 10,000 times, which, would also not run it time. So, we have to make another optimization of fast exponentiation. If we are raising the polynomial to an even power, x, recursively calculate the polynomial to the power x/2, and the square the result. This brings down the number of polynomial multiplications per test case down from 10,000 to about 28. ($2^{14} > 10,000$ and in the fast exponentiation we never do more than 2 multiplies for each power of 2 in the exponent.) While this is a lot of computation, it's within the bounds of the amount of computation required for harder problems in programming contests.

Once we've computed $[P(x)]^k$ with coefficients mod 997, we can quickly answer each query for that shopping spree just by accessing the appropriate coefficients in the result.

Some last details to keep in mind: FFT doesn't do mod, so we do all of our modding outside of the FFT. Although we only need to save coefficients up to 50,000 to store our final result, FFT requires twice as much room and a perfect power of 2 for the size of its input array. Thus, we need an array of size $2^{17} = 131,072$ to store our polynomials for this problem. Each time we run our FFT, we must "scale" our result. Scaling our result means two things: (1) zeroing out all terms above coefficient 50,000, since these don't figure into any of our final answers, (2) modding all terms by 997.