Florida High School Programming Series
2016 Playoff

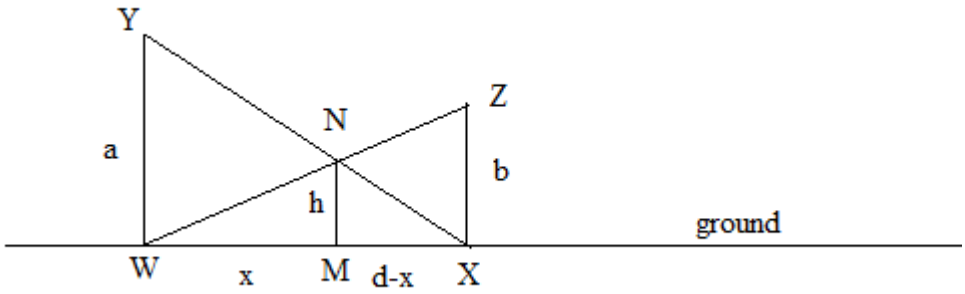
Problem Writer's Analysis by Arup Guha

Problem A: Host City

Upon first inspection, it might look like you have to use a special algorithm, but once you carefully read the problem, you realize that you are just looking for the cheapest city to host the grading event and that there is a straight-forward way to calculate the cost for hosting the event at each city. The key in calculating this cost is recognizing that you have to pay for each teacher for a particular city. Thus, if the total cost for a single teacher going from city A to city B is $1000 and there are 12 teachers in cityA, then we're adding a total of $12,000 for those 12 teachers if we host the conference in city B. (Also, there is no cost for the teachers who live in the host city already.) The run time of this solution (a double for loop) is $O(n^2)$, where n represents the number of input cities. Since $n \leq 20$, this is very easily fast enough.

Problem B: Flagpoles

Use the drawing given in the picture, adding a single variable x, and labeling the vertices we have in the diagram we have:



The key here is to notice that XYW and XNM are similar triangle and that WZX and ZNM are similar triangles. Once we have these two sets of similar triangles set up, we can set up several important ratios based on the tangent of angles YXW and ZWX:

$$\frac{h}{x} = \frac{b}{d} \qquad\qquad \frac{h}{d-x} = \frac{a}{d}$$

Using the first equation we find that $h = \frac{bx}{d}$. Now, substitute that into the second equation:

$$\frac{bx/_d}{d-x} = \frac{a}{d}$$
$$\frac{bx}{d(d-x)} = \frac{a}{d}$$
$$\frac{bx}{(d-x)} = a$$

$$bx = a(d - x)$$
$$bx = ad - ax$$
$$ax + bx = ad$$
$$x(a + b) = ad$$
$$x = \frac{ad}{a + b}$$

It follows that $h = \frac{bx}{d} = \frac{bad}{d(a+b)} = \frac{ab}{a+b}$.

Problem C: Ordering Paper

This is one of the easier problems in the set. The key is realizing how many sheets of the large paper are needed, given the number of pages an exam is. A single sheet of large paper can be used for 4 exam pages. Mathematically, our answer is $s = \left\lceil \frac{p}{4} \right\rceil$, where s is the number of sheets we need and p is the number of pages in the exam. The upside down 'U' symbol is the ceiling function. In particular $\lceil x \rceil$ is the least integer greater than or equal to x. (This way, if there is any leftover, we automatically get a full sheet more of paper.) In code, we can either handle this with an if statement, adding 1 page if s%p isn't 0, or we can make use of integer division setting the number of sheets to (p+3)/4. In general, in code, given two positive integers x and y, the value $\left\lceil \frac{x}{y} \right\rceil$ can be represented as (x+y-1)/y. Once we have this down, the rest of the problem is fairly simple, for each exam, first calculate the number of sheets of paper for that exam, then multiply that by the number of students. Finally, add up all of these products over all of the exams.

Problem D: Pass or Fail

This is the easiest problem in the set, just requiring an if statement inside of the loop that processes all the cases. I bet the students wish there were AP exam questions this easy =)

Problem E: Rating Teachers

Each year I try to put a custom sorting problem in the set, since I want to promote students learning how to custom sort as it's a very useful task in many situations. The key is creating an appropriate comparison function between teachers, as well as recognizing whether a teacher teaches a small class, a medium class or a large class. As the data is being processed, each teacher must be placed in one of three lists, based on their class size. In the comparison function, the passing percentage should be analyzed first. To avoid floating point errors, it's best to do the comparison via integers instead of between two floating point divisions. Say that teacher 1 had a out of b students pass and teacher 2 had c out of d students pass. If ad > bc, then teacher 1 had a higher pass rate. If ad < bc, teacher 2 had a higher pass rate. If those two products are equal, we move onto our tiebreaker, average exam score. Note that the input is small enough (a, b, c and d are all 100 or less), that no overflows will occur when you multiply.

For the tiebreaker, we must simply sum up the students' scores for a single teacher and divide by the number of students. To do this, we just multiply the number of students who got a particular exam score by that score, adding over all possible exam scores:

```
for (int i=1; i<scores.length; i++) {
     total += scores[i];
     sumScores += (i*scores[i]);
}
```

After this loop, we can calculate the average by looking at the ratio sumScores/total. Again, when doing our comparison, instead of using floating point operations, we can cross multiply and stick with integers. (Though, since the input guarantees no ties, the floating point division comparison is safe due to the small bounds on the input.)

Problem F: Correct Answer Recovery

This problem was supposed to be the hardest in the set, yet it was solved so quickly!

Since each test has at most 15 questions, there are at most 2^{15} sets of possible answers. So, what we can do is "try" each of these as the actual answers to the test, and see if they are consistent with the scores each student in the class received. To check for consistency, we take each student's set of answers and score them against the supposed key. Then we take that score and compare it to the actual score the student received. Finally, if we see that these numbers don't match, then we have proof that the answer key we were testing couldn't be the actual answer key, so we stop trying it and move onto the next possible answer key. But, we don't have what each student scored, so what we do is calculate how many each student would have gotten correct with each key and see if the that frequency chart matches the actual result frequencies.

One note is that if we use an integer bitmask to store the true/false answers (so, for example, 101110 for a 6 question test scores the answers TFTTTF), then if we do a bitwise XOR between the supposed answer key and a student's response, the result will have a 1 (T) in each bit spot where the student missed the question. For example, using the key above, if the students responses were 001100, then taking the bitwise XOR we get 101110 ^ 001100 = 100010. The number of bits that are one in this integer represent the number of questions the student missed. In Java there is a method, Integer.bitCount, which precisely returns the number of bits set to 1 in the binary representation of a number. Thus, an elegant implementation of this problem is to store each student's score as a bitmask, and then do a for loop through each possible answer key (just an integer bit mask), trying each one out. To try out an answer yet, just make use of the bitwise xor and Integer.bitCount method and voila!

Here is the segment of my code which make the critical calculations:

```
for (int i=0; i<(1<<numQuestions); i++) {
    int[] thisFreq = new int[numQuestions+1];
    for (int j=0; j<numStudents; j++)
        thisFreq[numQuestions-Integer.bitCount(i^answers[j])]++;
    res += possible(thisFreq, freq);
}
```

Problem G: Sorting Exams

It turns out that the solution to this problem is a greedy one. Whenever we merge two stacks, we always want to merge the two smallest stacks that remain. The formal proof involves something called the "exchange principle." Taking any algorithm where two non-minimal stacks get merged first can be equalled or improved by just merging the two minimal stacks first. We can see this in the 3 stack case. Consider three stacks of papers a, b and c with a < b < c. No matter what our first merge is, our last merge will take a + b + c time. Given that the last merge is the same in all situations, we seek to minimize the work of the first merge, which is clearly done by picking stacks a and b. This logical can then be used inductively (if this fact is true for 3 stacks, we can show it's true for 4 stacks, and so forth.)

So, simply take the data and find the two smallest values. Add them, add this sum to the total operation count, and then put this sum back into the list. Repeat until there is only 1 number left, representing the final sorted stack. In terms of implementation, an efficient implementation of this would simply use a priority queue, which quickly finds the mininum in a list and quickly allows for inserts into that list. (Both operations take O(lg n), where n is the total number of items in the list.) The data structure used to implement a Priority Queue is a heap. Both Java and C++ have implementations one can use so that they don't have to implement their own.

The input data is small enough however, that no priority queue is needed with n = 500, we can still easily run an $O(n^2)$ algorithm where we look for the minimum in the list using a loop through all of the data.

Problem H: What to Teach?

This problem is precisely the 0-1 Knapsack problem. The class time are the weights from the 0-1 Knapsack problem and the points on the exam for each class are the values from the 0-1 Knapsack problem. Here is the Wikipedia page on the 0-1 Knapsack problem:

https://en.wikipedia.org/wiki/Knapsack_problem

In short, we create an array such that dp[i] stores the most valuable knapsack with weight i. These are all initially set to 0. When considering an item with weight j and value v, we arrive at the following for any new knapsack of weight i:

```
dp[i] = max(dp[i], dp[i-j]+v)
```

Basically, if we don't take this item, our best knapsack value remains the same. If we do take this item, our new value is the value of the remaining knapsack with weight i - j, plus the value of this item. Thus, we must simply make this comparison over all items and all weights. To make sure we don't allow the use of an item more than once, our loop through all distinct weights must run backwards. The runtime of this algorithm is O(nW), where W is the maximum weight. Here is my code which implements this logic:

```
int[] dp = new int[totalMin+1];
for (int i=0; i<n; i++)
    for (int j=totalMin; j>=times[i]; j--)
        dp[j] = Math.max(dp[j], dp[j-times[i]] + pts[i]);
```