# Lecture Notes, CSE 232, Fall 2014 Semester

Dr. Brett Olsen

## Week 13 - Problem Analysis

So, the very first week of class, I gave sort of a general strategy for programming contests — how to pick which problem to solve, general goals for solving problems, and so on. Then for all of the following weeks, we covered lots of different kinds of problems you might run into and how to attack them. This week, we're going to go back to that strategy we started off with — rather than talk about new algorithms or approaches to solving problems, we're going to do some practical work.

I'll go over a number of different problems and discuss how you might approach a new problem during a contest. The homeworks I've given you so far have always been related to the topic we discussed that week, giving you a headstart on finding a correct approach. Normally, though, you'll need to analyze the problem and first identify how you might solve it, only then moving on to correctly implementing a solution. During the class this semester, we covered a lot of different approaches and tools for solving problems:

- Brute Force (Complete Search)
- Greedy (pick the locally optimal choice and hope it leads to the globally optimal solution)
- Dynamic Programming (exploit repeated subproblems)
- Data Structures (stacks, queues, heaps, dictionaries, suffix trees, bitmasks, etc.)
- Graphs (graph construction, traversal, connectivity, shortest paths, etc.)
- Computational Geometry (convex hull, area of a polygon, basic geometric analyses)
- Combinatorics (Fibonacci, binomial, Catalan numbers — are we asked to count things?)
- Number Theory (primes & prime factorization, modular arithmetic)
- Random Numbers (generating random numbers, using randomness to find solutions)

What I want to talk about this week is how to choose an approach to each problem we're presented with. To do this, here's some questions to ask yourself about each problem that will be helpful in picking an approach.

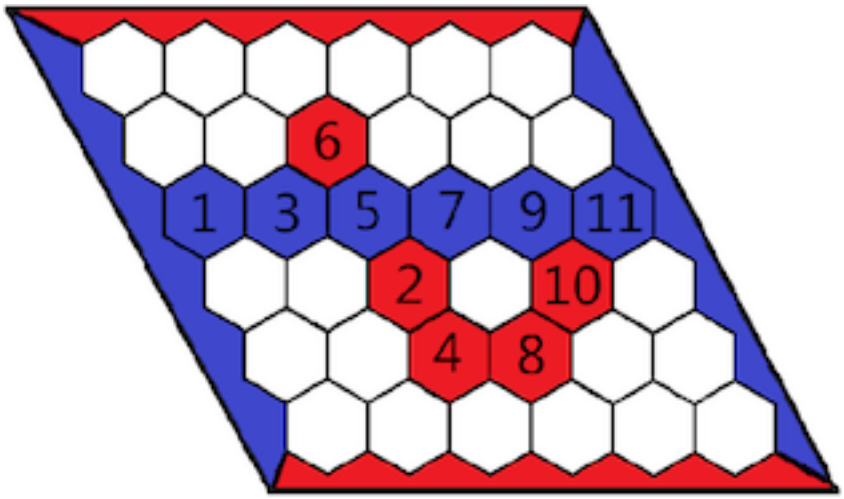
- Do you want *the* correct answer or *a* correct answer? (If there's lots of correct answers, it can be much easier!)
- How many possible answers could there be? (Is a complete search feasible?)
- Can you break the problem down into smaller subproblems? (Necessary for greedy or dynamic programming solutions.)
- Can we greedily combine the solution to smaller subproblems to solve the larger problem? (Greedy solution?)
- Do we see the same subproblems repeatedly? (Dynamic programming!)
- What operations will we need to perform repeatedly? Is there a data structure that efficiently supports these operations?
- Does the problem concern itself with connections between elements? Can we model it as a graph, and are there known graph algorithms that can find the solution?
- Does the problem ask us to count something? Is it represented by a well-known combinatoric sequence? (Fibonacci, binomial, Catalan numbers)
- Does the problem deal with large integers where prime factors or modular arithmetic could be useful?
- Can we use any known libraries in our language to simplify parts of the solution?

**Problem 1 - Hex** https://code.google.com/codejam/contest/2929486/dashboard#s=p2

The first problem I want to discuss is Hex, from the Google Code Jam China New Grad Test this year.

Hex is a board game, played on an N by N board, with each cell being a hexagon, connected to six of its neighbors. The upper and lower sides of the board are marked red, while the left and right are marked blue. Red and blue players take turns placing stones of their color on an empty cell, with the winner of the game the first player to form a path of their color of stones from the sides of the board of their color.

```
In [1]: from IPython.display import Image, display
        display(Image(filename='./week13_notes_img1.png'))
```




The question we're asked is, given the board and positions of pieces on the board, what's the state of the board: does Red win, or Blue win, or has nobody won yet, or is the board state impossible to reach?

Let's go through that list of questions and see if we can find the right approach to take with this problem. To see if red or blue won, we'll need to look for connected paths between the left and right (for blue) and top and bottom (for red) sides of the board using pieces of the right color. That phrase "connected paths" ought to be a hint to you to try modeling the problem as a graph. As two graphs, in fact — one for blue and one for red. Each cell containing a blue piece is connected to any neighboring blue pieces, along with the neighboring left/right edges. Similarly for red. Once we've built the blue and red graphs (including the board edges as vertices!) we can use them to determine whether the left/right edges are connected (and blue has a connected path) or whether the top/bottom edges are connected (and red has a connected path). So we're done, right? If blue has a connected path, blue wins, if red has a connected path, red wins, and if nobody has a connected path, then nobody wins.

But what about the impossible board states? We'll need to check for those as well. There's several ways the board state can be impossible, and it takes some thinking to identify all of them:

- The difference between the number of red and blue pieces on the board is more than 1 (somebody skipped a turn)
- The winner of the game has fewer pieces on the board (their opponent placed a piece after the game was over)
- The winning path between board edges can't be broken by removing a single piece (or in graph terms, has no *articulation vertex*) (the players kept playing after the game was over)

**Problem 2 - Password Attacker** https://code.google.com/codejam/contest/4214486/dashboard

The second problem I want to look at is called Password Attacker, and it involves trying to guess a password given some limited information about that password. In particular, we know that the password has $N$ total characters and $M$ unique characters, with $M \leq N$. We're asked to give the total number of

possible passwords they might be, modulo $10^9 + 7$. Before we get into how to actually solve the problem, let's think about how to classify it. Just like in the last problem, where the phrase "connected path" ought to have tipped you off that we were looking at a graph problem, the phrase "how many possible passwords" ought to tell you that this is a combinatorics problem. It is not, however, a *simple* combinatorics problem. So let's think about a simpler case, where $M = N$. That means that no character is repeated in the password, so the number of possible passwords is simply the number of permutations of size $M$, which is $M!$.

Now, how might we extend this where $M < N$? Well, suppose we had a rule that as we build up our passwords starting from the beginning, we always add new characters in lexicographic order. So for $M = N$, we'll always get the string "abcd..." of how ever many characters. For $M < N$, though, we'll have multiple possiblilities. For M=2, N=3, for example, we'll have the passwords "aab", "aba", "abb". To get the total number of possible passwords, we'll have to swap those characters around, in every possible way. *This is exactly equivalent to a permutation.* So, we can multiply the number of passwords following this rule by $M!$ to get the total number of passwords.

How can we generate these lexicographically ordered passwords? Well, at each character position, we have two options: repeat one of the characters we already have, or add a new character. Now, sometimes, one of those options might be foreclosed. If M=3 and N=4, then we're only allowed one repeated character, so once we've repeated a character we can't ever do it again. Similarly, if we've already added $M$ new characters, then we're out and we'll have to repeat characters until we've generated $N$ total characters. That suggests we keep track of two variables to define a state: the number of repeated characters that are allowed in the rest of the password ($R$) and the number of unique characters we've already used ($C$).

Then the initial state, once we've added the first character is $(R = N - M, C = 1)$. We can define a recurrence as $(R, C) = C * (R - 1, C) + (R, C + 1)$. The base cases happen for $R = 0$, in which there is only one way to generate the rest of the password, and for $C > M$, in which there are zero ways to generate such a password. So we write the recursion, try to solve the problem, and find that it runs far too slow on the Large input. What trick can we use here? We know that $R$ can vary from $N - M$ to 0, and $C$ can vary from 1 to $M$, where $M$ and $N$ can be as large as 100, so there are no more than 10000 distinct calls for each problem. The slow run time comes because we have *overlapping subproblems*, a hallmark of dynamic programming. So by either implementing an array to store values or simply memoizing our recursive function, we can solve the problem very rapidly.

*Aside: If you're familiar with combinatorics, you might recognize our recurrence as the Stirling numbers of the second kind. This knowledge would make it easier to quickly implement your algorithm!*

**Problem 3 - Kingdom Rush** https://code.google.com/codejam/contest/1645485/dashboard#s=p1&a=1

The third problem is called Kingdom Rush, based on a little mobile tower defense game. This game has lots of levels ($N$), and completing a level can earn you either one or two stars, depending on how well you do. Stars make you more powerful, so that it will be easier for you to complete future levels. You can only gain the stars from each level once, but if you complete a level with a one-star rating, you can go back later once you're more powerful and complete it with a two-star rating to earn the second star. You already know how many stars you will have to have in order to be powerful enough to complete each level with either a one- or a two-star rating. You want to know what's the minimum number of times you have to complete levels in order to finish all the levels with a two-star rating. This could be as low as $N$ (if you can complete every level once with a two-star rating) or as high as $2N$ (if you must complete every level twice, first with a one-star rating and then with a two-star rating) or it might not be possible to finish all the levels at all.

How might we answer this problem? What kind of approaches can we use? Well, we can define a state as the number of stars we've earned, the level completion count, and the list of all the levels we haven't yet completed. One approach would be a complete search - at each state, we go through each level it's possible to complete and continue our search from that state, ending when all the levels are finished. Then we return the minimum completion count from all the finished states. In the Large case, there's 1000 levels, giving us up to 1000! permutations of the order in which to try levels — far too big to run. We can notice that the problem exhibits optimal substructure - the result for each state depends on the result of smaller substates, and we can solve the smaller substates independently. So we might think about a dynamic programming approach - we can have from 0 and 2000 stars, but there are $3^{2000}$ possible subsets of levels still to be completed. This is still too big to evaluate.

What happens if we try to identify the "best" move at each state? Well, first off, let's notice that the

best thing to do for each level is to complete it once with a two-star rating. That suggests that if, at any time, we have enough stars to complete a new level with two stars, we can just go ahead and do so right now. Waiting until we have more stars won't improve our performance on that level; it's already as good as it can be. That means taking this move can never be a bad idea, so we don't have to examine any other possible moves before taking it.

What if there are no levels we can complete right now with two-star ratings, but there are levels we can complete with a one-star rating? We'll have to pick one of them to complete, but which one? Suppose there are two remaining levels. We have enough stars to complete both levels with a one-star rating, but to earn a two-star rating requires one additional star for the first level and three additional stars for the second level. If we complete the first level with a one-star rating, we earn an additional star. We still can't complete the second level with a two-star rating, so we complete the first level again with a two-star rating. Now we have to run through the second level twice to finish it, for a total of four levels completed. But if we do the second level with a one-star rating first and earn an additional star, we now have enough to do the first level with a two-star rating without having to do it as a one-star rating first, letting us finish the game with only three additional levels completed.

That suggest that when we have a choice between multiple levels which we can only complete with one star, we should choose to complete the one with the highest requirement to complete with two stars, because that additional star might enable us to two-star some of the other levels.

Together, this gives us a complete *greedy* strategy, which only requires $O(2N)$ time to compute:

1) First, complete any levels which we can two-star that we haven't yet done.
2) Then, complete any level we can one-star with the highest requirement to two-star.
3) Repeat until all levels are completed with two-stars. If we ever end up with no possible levels to complete, then we can't win the game.

**Lab Section** No homework problem this week! Instead, during lab section, I'd like you to look at the following problems and try to identify approaches that might be used to solve them — no actual solutions required.

- Problem 1 - Copying Books - There are several potential approaches to take here.
- Problem 2 - Osmos
- Problem 3 - Parentheses Order - This one is hard!