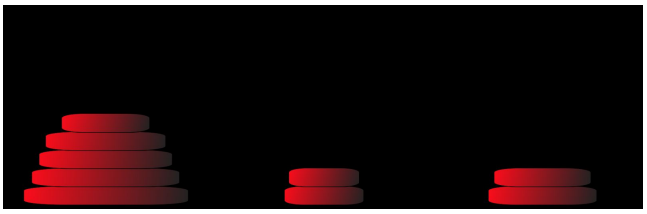
Excusez-moi. I don't mean to annoy
But how do you goa from hare to hanoi?

How do you go...

... from hare to Hanoi?

In this quest you get to take recursion to the next level. For those of you in my CS2A class from before, think back to a time when we talked about recursion, Fibonacci, memoization and dynamic programming[1]. That's what you're gonna do in this relatively easy quest.

Well, almost. Except the dynamic programming part. But I will be happy to see a thoughtful discussion in the forums on exactly what the trade-offs are between memoized recursion and DP, how a table-driven approach may work, and whether such an approach gives any real wins. Why (if) or why not (if not).



Overview

The problem statement is this: You have N discs of decreasing radius and three poles[2] on which these discs could be slotted. You can only move one disc at a time, and you cannot place a bigger disc on a smaller one. Suppose all N discs are on pole 1, what is the sequence of moves you need to transfer all discs to pole 2? Assume that you can use the third pole, 3, as your temporary holding space. You can use it to store any number of discs, as long as you don't violate the disc ordering property (a bigger disc may not rest on a smaller one).

You are welcome to try out the recursive (but not memoized) Javascript demo on my home page. Work it out on a piece of paper for 1 disc, 2 discs, and 3 discs. Then work out a general technique for N discs. For example, suppose I denote the action of moving the top disc on pole 1 to pole 2 (if valid) by the string "`1->2`". Then I might say:

1. To move 1 disc from 1 to 2, I need to do: `1->2`
2. To move 2 discs from 1 to 2, I need to do: `1->3, 1->2, 3->2`
3. To move 3 discs from 1 to 2, I need to do: `etc.`
4. ...
5. To move N discs from 1 to 2, I need to: ...

[1] Darshan recorded it and posted it on our youtube class channel (nonlinearmedia). See if you can locate it. Please share the link in the forums if you think it could be useful to this quest.

[2] I don't remember why I used the word *poles*. Maybe I was thinking of slotted discs in some actual puzzle I had seen. We'll continue to use that word, but we just need 3 places to rest the discs - that's all.

Your challenge is to discover the pattern in the sequence above and code it up, so I can ask your program to give me the move sequence for any number of discs. Of course, I'll keep the number reasonable. But I don't know what that value is right now.

There are two parts to this quest. The first part is easy. It's just straight recursion. For the second part, you're going to have to memoize the first part. Here is the mostly-filled-in `Hanoi` class you need to put in your `Hanoi.h`:

```
class Hanoi {
private:
    int _num_poles;
    int _num_discs;

    // TODO: Declare the _cache member using an appropriate
    // level of nesting within std::vectors to put each string
    // of moves. You should be able to access the cache like so:
    //   _cache[num_discs][src][dst] = "move1\nmove2\n..."

    std::string lookup_moves(int num_discs, int src, int dst);
    std::string get_moves(int num_discs, int src, int dst, int tmp);

public:
    // Use freebie default constructor
    std::string solve(int num_discs, int src, int dst, int tmp);

    friend class Tests; // Don't remove this line
};
```

You need to supply the implementation file, `Hanoi.cpp`, yourself. The following miniquests will shed more light on the individual methods.

Your first miniquest - Basis Cases

All you need to do to pass this miniquest is to return the right response for your basis cases. These are the situations where you return a result immediately without recursing deeper. You can be defensive and define two basis catches: for num_discs = 1 and num_discs = 0.

For each case, you can imagine 6 different puzzles - yes? For example, my problem might be to move 1 disc from pole 1 to pole 2. Or it might be from pole 3 to pole 1. How many such cases can you imagine? Your `get_moves()` method must return the sequence of required moves for each case as a series of newline separated strings of the form `A->B` (It must be a proper ASCII hyphen followed by a greater-than sign. Don't copy and paste these symbols as your output will not be processed before being checked.

Of course, you won't have more than one move in the basis cases. So there is only ever a maximum of one newline character at the end (none for 0). But in general, you can expect to

return a sequence of moves. For example, to move 2 discs from 1 to 2, you would return the string `"1->3\n1->2\n3->2\n"`

Your second miniquest - Get moves

Make sure your `get_moves()` method is now able to handle input for cases other than the basis case. E.g. you may want to find the sequence of moves to transfer 12 discs from pole 2 to pole 1.

That's all. If your recursion is clean and tight, you'll find your method is no longer than 5-10 lines of code. However, remember that you haven't built in memoization yet (you can expect the method to grow to about 15-20 lines after it's been memoized.

Your third miniquest - Solve

A piece of cake. This method should simply return the result of a `get_moves()` call prefixed with the following line:

```
# Below, 'A->B' means 'move the top disc on pole A to pole B'
```

Note that there are no spaces before or after the first and last visible characters. The quotes used in the string are plain ASCII single quotes. Don't use curly or other fancy quotes which might creep in if you copy/paste from non-plain-text sources.

Your fourth miniquest - Memoize

The real challenge in this miniquest is in implementing and managing the data structure for the cache correctly. The abstract representation of the cache should look like in Figure 1.

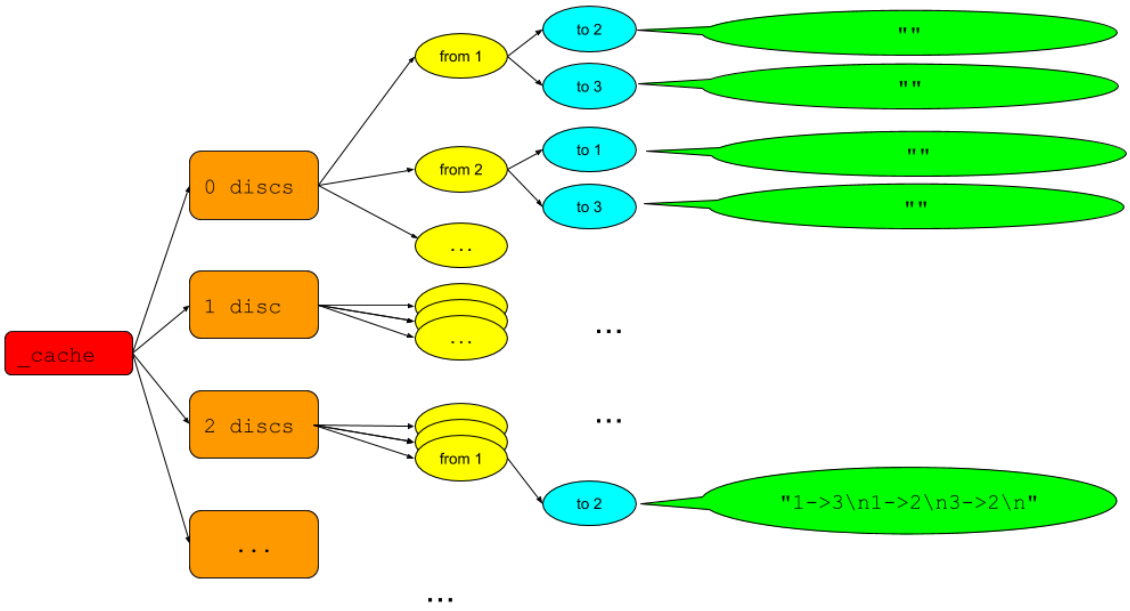


Figure 1: Cache of Hanoi

To get this functionality, implement the cache as a `vector` of `vector` of `vector` of `strings`, using Figure 1 as a guide in understanding the semantics of the vector. For example, the second index, `j`, in the sequence of nested indexes in `_cache[n][j][k]` stands for the source pole number to solve a puzzle with `n` discs.

Note that the above pole numbers span values 1, 2 and 3, but your cache is a `vector` whose elements start at `n=0`. If you look at our implementation, it doesn't really matter what these numbers are - just that they are different. However, there is a relatively tiny bit of wastage you will incur if you use large valued pole numbers (e.g. 100, 150, 200). Exactly how much wastage? Why do you think we can consider it tiny? relatively? Or can we? (Discuss)

Important: Use pole labels 1, 2 and 3 (not 0, 1 and 2 or any other). This means that `_cache[n][0]` and `_cache[n][j][0]` are *dead locations* you won't use in your program. But the savings you get in reduced cognitive load (from a mismatch between emitted pole numbers in the output and vector indices) will make it a good deal in this quest.

In order to ace this miniquest, you need to do the following correctly. It's all or nothing, I'm afraid:

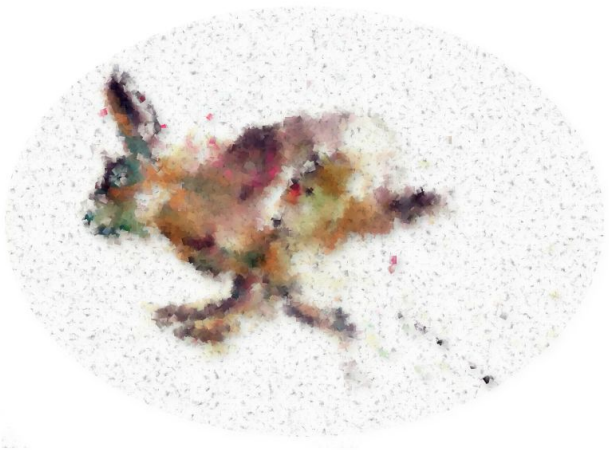
1. Declare the `_cache` object in your Hanoi class
2. Implement the method `lookup_moves()` which should
    a. look up the cache for a given triplet (num_discs, source, target)
    b. Return the cached string if it exists
    c. Else return the empty string ""
3. Edit the method `get_moves(),` which should return the string of moves required to transfer `num_discs` discs from the `src` pole to the `dst` pole using the `tmp` pole as a holding area. I might invoke it like: `your_hanoi.get_moves(5, 2, 3, 1)` to get the moves to transfer 5 discs from pole 2 to pole 3 using pole 1 for holding.
4. You should not hold on to any `_cache` entry longer than needed for a 1-time calculation with a given number of discs (think like you need to calculate `fibonacci(n)`)
5. You should only create _cache nodes lazily. That is, don't create _cache[i][j] until such time as you actually have something to put in it.

How do you need to change `get_moves()`?

- When a request for a move sequence arrives, check the cache first using the lookup method.
- If a cache entry exists, return it.
- Otherwise, calculate the sequence of moves as before, but store it in the cache before returning it to the user.
- **Seriously** think about whether you need to always carry your whole cache with you.

This is a tricky miniquest to get right and it is worth a significant number of points. But a correct implementation here will give you solid insight into how to make a table-driven algorithm that does the same thing as memoized recursion (except with no recursion at all). I'm not testing your table-driven version. But rewards may await interesting thoughts shared on this topic.

Other questions and worthy discussion topics are welcome too. Here's something for starters: How long does a cache entry need to live? How expensive is it to maintain?



Submission

When you think you're happy with your code and it passes all your own tests, it is time to see if it will also pass mine.

1. Head over to https://quests.nonlinearmedia.org
2. Enter the secret code for this quest in the box.
3. Drag and drop your `Hanoi.*` files into the button and press it.
4. Wait for me to complete my tests and report back (usually a minute or less).

Points and Extra Credit Opportunities

Extra credit points await well thought out and helpful discussions.

Happy hacking,

&