MTH 4300, Lecture 7

Binary Search;
Making Change;
Towers of Hanoi;
Pronounceable Words [not covered in class]

E Fink

February 25, 2025

1. Binary Search

Imagine that you have a SORTED array, and you wish to search for an entry within it. Here's an efficient strategy:

- Set your "search range" to start from the beginning of your array, and end at the end of your array.
- Compare your value of interest to the entry right in the middle of your search range.
- (Recursion!) If your value of interest is less than the midway value, perform a new search with the right boundary of your search range updated to one less than the midway position. If your value of interest is greater than the midway value, perform a new search with the left boundary of your search range updated to one more than the midway position.
- Return `true` if you ever find the value of interest; return `false` if your search range is empty.

L7x1_binary.cpp

2. Making Change

Fact I learned from a fun-fact display in a shop near my house: there are ??? ways to make change for a dollar, using pennies, nickels, dimes, quarters, half-dollar and dollar coins. (See the program from the answer.)

Here's how we can approach this recursively. The idea is to use a function with TWO parameters:

- The value `price` (in cents) you wish to make change for, and
- the HIGHEST coin value `top_coin_value` (in cents) you wish to use.

The function will return the number of ways to make change for `price` using coins no higher than `top_coin_value`.

The signature line for the function will be

```
int num_configs(int price, int top_coin_value)
```

Warm-up 1: what should `num_configs(23, 1)` return?
Just 1, since there is only one way to make 23 cents out of pennies. When the second argument is 1, that's a base case!

Making Change

Warm-up 2: what should `num_configs(23, 10)` return?

If dimes are the highest denomination available to us, we can separate our coin configurations into those which use at least 1 dime, and those which use 0 dimes.

How many configurations use at least 1 dime? After you use 1 dime, you still have 13 cents left over, and you have to count the number of ways to make change for 13 cents using dimes, nickels and pennies. In other words, the number of these configurations is `num_configs(13, 10)`.

How many configurations use no dimes? These configurations use only nickels and pennies. There are `num_configs(23, 5)` such configurations.

So, in total, there are `num_ways(13, 10)` + `num_ways(23, 5)` configurations. Those two are smaller problems!

How would you break down the following into sums of smaller problems?

In each line, the first summand is the number of configurations which have at least one of the top coin, and the second summand is the number of configurations which have none of the top coin.
`num_configs(57, 25)` = `num_configs(32, 25)` + `num_configs(57, 10)`
`num_configs(31, 5)` = `num_configs(26, 5)` + `num_configs(31, 1)`
`num_configs(19, 25)` = `num_configs(-6, 5)` + `num_configs(19, 10)`
I guess we should watch out for the first case, which should be 0.

Making Change

What have we learned?

The base cases should be when the highest denomination is pennies – there's one way to make any price out of pennies – and when the amount we wish to make change for is negative – there are zero ways to do that.

Otherwise: split the calculation into the number of configurations that contain at least one of the highest denomination, and the number of configurations that only use lower denominations. The former is gotten by counting the number of configurations when you remove one of the highest denomination from the price.

L7x2_change.cpp

Making Change

In **L7x3_helper.cpp** , we approach the problem in a slightly different way. The real question isn't

> *How many ways are there to make change for n cents, using coins less than a dollar?*

The question is, more succinctly,

> *How many ways are there to make change for n cents?*

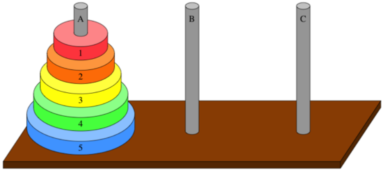
The second parameter in our function is necessary for our strategy, but it's a bit unattractive to make the original call to our function require an unnecessary parameter which is always 100.

In the new version, we make a one parameter function which counts change, but this function just calls a "helper" function, which performs the actual work.

3. Towers of Hanoi

This is one of the most famous recursive problems. There are *n* disks on the left pole, stacked in order of decreasing size. The goal is to move this stack to the right pole.

The rules: you can only move one disk at a time, and you can never place a disk on to of a smaller disk.



https://www.mathsisfun.com/games/towerofhanoi.html

Towers of Hanoi

How do we solve this? Here's a strategy:

- Move the smallest $n-1$ disks temporarily to the pole we're **not** targeting, leaving the biggest disk uncovered.
- Move the biggest disk from the original pole to the pole we **are** targeting.
- Move the smallest $n-1$ disks from the pole they're on to the target pole.

But how do we move the smallest $n-1$ disks? That's what recursion is for. (What would the base case be? The instructions only make sense when $n > 1$, so $n = 1$ ought to be the base case.)

Let's write a program that gives explicit instructions for the disk moves. Every instruction will be of the form

Move disk d from pole x to pole y.

L7x4_hanoi.cpp

Towers of Hanoi

One of the trickiest parts of this problem is understanding what the signature line of the function should be. I have written it as:

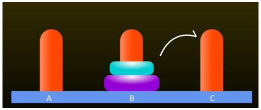
```
void towers(int num_d, char f, char t, char aux)
```

where `num_d` represents the number of disks we're trying to move; `f` represents the "from" pole (A, B or C) which we're moving the disks from, and `t` represents the pole which we're moving disks to. And `aux` represents the third pole, which can be used for temporary storage.

For example, for this picture, I would call

```
towers(2, 'B', 'C', 'A')
```

and it would print



```
Move disk 1 from pole B to pole A.
Move disk 2 from pole B to pole C.
Move disk 1 from pole A to pole C.
```

4. Pronounceable Words [not covered in class]

Let's say that a string of letters is *pronounceable* if it never has *more than two consonants in a row*. (AEIOU are vowels, the other 21 letters are consonants.)

How many pronounceable strings of length n are there?

Once again, let's think about how recursion can help us. In sequence problems like this, it's helpful to think about how the sequences start, and how they can continue. For this purpose, it will help to define the *head* of a string to be its first character, and the *tail* to be all the rest of the characters.

For example, the head of `hamburger` would be `h`

and the tail would be `amburger`

and the tail of the tail would be `mburger`.

Pronounceable Words

What can we say about the head and tail of a pronounceable string of length n?

- The head can be a vowel – there are 5 of these. In that case, as long as the tail doesn't have two consonants in a row, the whole string will be pronounceable. In other words, if the tail is pronounceable, then the whole string is pronounceable. So the total number of pronounceable strings starting with a vowel would be 5 times the number of pronounceable strings of length $n-1$.
- The head can be a consonant – there are 21 of these. It's a little trickier to describe what we need from the tail now: in addition to it being pronounceable, we need the tail to have at most one leading consonant.

The first count is natural to deal with recursively. How can we deal with the second count?

Pronounceable Words

Answer: broaden the function! Let's write a function

`int helper(int length, int mlc)`

which counts *the number of pronounceable strings which start with AT MOST* `mlc` *leading consonants*.

Base cases: you can't have a negative number of leading consonants – there are zero strings like that. Also, as long as the number of leading consonants isn't negative, the number of strings of length zero is 1 – the empty string.

L7x5_pro.cpp