

CSCI 132 Data Structures—Lab #6

Introduction

Today we will get some practice writing recursive functions, and think a bit about algorithm efficiency.

Setup: Login and copy the code for the lab using: `cp -r ~csci132/labs/lab6 labs/ && cd labs/lab6/`

Exponentiation

In this first part we will compare different ways we can write our own exponentiation function in C++, i.e., a function that computes x^n for some $n \geq 0$. Your task is to complete the implementation for the three functions described below in the file `Exponentiation.cpp`.

1. Complete the implementation of the function `power1` to compute x^n for $n \geq 0$ using a loop.
2. Complete the implementation of the function `power2` to compute x^n by using the following recursive formulation.

$$x^n = \begin{cases} 1 & n = 0 \\ x * x^{n-1} & n > 0 \end{cases}$$

3. Complete the implementation of the function `power3` to compute x^n by using the following recursive formulation.

$$x^n = \begin{cases} 1 & n = 0 \\ (x^{n/2})^2 & n \text{ is even} \\ x * (x^{n/2})^2 & n > 0 \text{ and } n \text{ is odd} \end{cases}$$

Implementation: Exponentiation can result in very large numbers and your code will likely result in overflow for regular conditions. For example 2^{100} is a 31 digit integer and will definitely not fit in a regular `int`. So, we have included a `constant int M` for which your code will have to return $x^n \pmod{M}$, which is the remainder x^n has when divided by M . For example, if $x = 2$ and $n = 32$, $x^n = 4,294,967,296$, but your code will have to return $x^n \pmod{M}$ for $M = 10007$, which is 2924. To achieve this, for each implementation of the function, whenever you are updating results using a multiplication, use a `% M` operation. For example:

`results = results * x` should be converted to `results = (results * x) % M`.

Next, note that to square $x^{n/2}$ in the implementation of `power3` you should multiply the result with itself. Your implementation of each of the above recursive functions should avoid unnecessary recursive calls, e.g., to square the result of a recursive call you should first save the result in a variable and multiply that variable with itself to compute the square.

Compile and test your three functions using the command in `Makefile`, or you can compile using the shortcut `make exponent`. Run the program using `./exp <power1/power2/power3> <base> <exponent>` where the first commandline argument is your choice of function, and the last two are the base and exponent of the expression.

Example: `./exp power1 2 1000`

After compiling and testing your code, answer the following questions.

Question 1: How many multiplications will your implementation of each of the functions `power1`, `power2`, and `power3` perform when computing 3^{32} ? And how many multiplications will they perform when computing 3^{19} ?

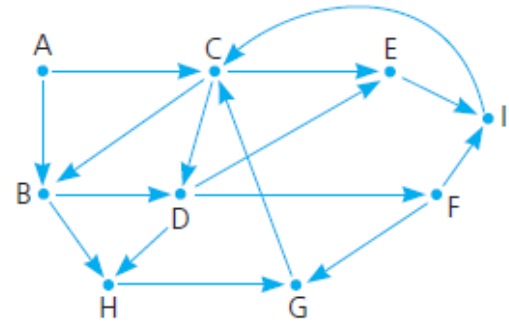
Question 2: How many recursive calls will your implementations of `power2` and `power3` make when computing 3^{32} ? And how many recursive calls will they make when computing 3^{19} ?

Question 3: Computing exponentiations for expressions with very large exponents with thousands of digits is important for applications in cryptography. To get a better idea of how our approaches compare, time how long it takes to compute the exponentiation of an expression with a large exponent with at least 9 digits (i.e., $\geq 100,000,000$) by running your program with the `time` command. For example, you can see how long it takes to compute $3^{100,000,254}$ using `power1` with the command `time ./exp power1 3 1000254`.

Record the base, exponent, and corresponding time printed for `power1` and `power3` in your `lab6.txt` file.¹

Flight

Given a map of cities, and flights between the cities, a traveler would like to find a *path* from one city to another. Consider the flight map to the right. There are nine cities, named *A* through *I*, and each arrow shows a direct, one-way flight from one city to another. If asked to find a path from city *B* to city *C*, one solution is the path *B* → *H* → *G* → *C*. Another solution is the path *B* → *D* → *F* → *G* → *C*. Sometimes there are no solutions. For example, there are no paths from *D* to *A*. Note: the arrows represent one-way flights, and can't be used to go in the opposite direction.



Algorithm: Recursive backtracking can be used to solve this kind of path-finding problem. The algorithm recursively builds up a partial solution—an itinerary consisting of list of cities in the path the traveler should take, beginning with the starting city and with each pair in the list connected by a direct flight. Each recursive call adds a new city to the itinerary, hoping to eventually reach the destination city. At each point in the algorithm, given a partial solution, there could be several possible neighboring cities the traveler could go to next. While trying to find a path from *D* to *G*, for example, if the (partial solution) itinerary so far is *D* → *F* → *I* → *C*, then as a next step the traveler could depart city *C* and fly directly (following the arrows leaving *C*) to either *B*, *D*, or *E*. The algorithm should try each of these in turn: add a city to the itinerary, recursively check if that leads to a solution and, if not, remove that city and try a different one. The algorithm should only give up if none of the possible next steps leads to a solution.

Loops: It is unreasonable for a flight path to contain loops. We also don't want our backtracking algorithm to get stuck in an infinite recursion, adding cities to an itinerary that don't actually get any closer to the destination but instead make the traveler go around and around in circles. In the previous example, when the (partial solution) itinerary was *D* → *F* → *I* → *C*, and the algorithm is considering a choice of *B*, *D*, or *E* to add to the itinerary, we should *not* send the traveler to *D*, since that would cause a loop in their itinerary. Similarly, if for a potential next-step city, like *E*, that is *not yet* included in the current partial solution, we might want to avoid that city if it was already visited during some previous failed attempt to find a path.

Complete the code in `Flight.cpp` to implement the above backtracking algorithm. The recursive function you are to write, `findPath()` takes several parameters:

- The names of two cities, `current` and `dest`, representing the city where the traveler is currently located, and the destination the traveler is trying to reach. Initially, when the `main` function calls your function, it will pass the starting city name for `current`.
- A `flightMap` containing information about all available direct flights. This is an array, where each element of the array is an `ArrayList` of information about a single city. A city's `ArrayList` has the list of next-hop cities that are directly reachable from it. Note: Arrays are indexed by integers,

¹If you attempt to compute the result of an exponentiation with a large exponent using `power2` you will receive a `Segmentation Fault`. This is due to the very large number of recursive calls done by this approach for large exponents.

but city names are strings. Use the provided `getCityNumber()` function to convert the name to an integer. For example `flightMap[getCityNumber("D")].getCurrentSize()` will give you the number of flights leaving city *D*, and `flightMap[getCityNumber("D")].getEntry(i)` will give you the name of the i^{th} city that can be reached from *D* by direct flight.

- **visited** a set of names of cities the visitor has already visited. This is not a `const` parameter, as it is expected the algorithm will add names to this set as it tries the various cities in turn during each step. These cities can also be avoided in future steps, to avoid going in circles.
- **solution** is the partial solution representing the current itinerary being worked on by the algorithm. It is also used to hold the actual solution, if a path is eventually found. This is not a `const` parameter either, as your function is expected to add cities to this and, if successful in finding a path, leave the solution found in this variable so the `main` function can print it.

Compile and test your code using the command in `Makefile`, or you can compile using the shortcut `make flight` then run the program using `./flight` (you can also put the start and destination city names right on the command line).

Output:

- Your code should output the current partial solution any time that they add a new city in search of the final solution.
- At the end, you should print out whether there is a solution or not. If there is a solution, print out the full path.

Example output:

```
>> ./flight A C
```

```
Searching for a path from A (city #0) to C (city #2)...
Traveler is at A trying to reach C with itinerary so far: A
Traveler is at B trying to reach C with itinerary so far: A -> B
Traveler is at H trying to reach C with itinerary so far: A -> B -> H
Traveler is at G trying to reach C with itinerary so far: A -> B -> H -> G
Path found: A -> B -> H -> G -> C
```

```
>> ./flight D A
```

```
Searching for a path from D (city #3) to A (city #0)...
Traveler is at D trying to reach A with itinerary so far: D
Traveler is at H trying to reach A with itinerary so far: D -> H
Traveler is at G trying to reach A with itinerary so far: D -> H -> G
Traveler is at C trying to reach A with itinerary so far: D -> H -> G -> C
Traveler is at B trying to reach A with itinerary so far: D -> H -> G -> C -> B
Traveler is at E trying to reach A with itinerary so far: D -> H -> G -> C -> E
Traveler is at I trying to reach A with itinerary so far: D -> H -> G -> C -> E -> I
Traveler is at F trying to reach A with itinerary so far: D -> F
Sorry, you can't get from D to A.
```

Hints and Suggestions:

- Think about what the base case (or cases) are, and when a function will need to recursively call itself.
- The return value is important. Not only does `main` want the correct return value, it should be used to check if a recursive call has succeeded in finding a solution or not.

- You may find it useful for debugging purposes to print the partial solutions at every step of the algorithm. Use:
`cout << solution.toStringWithSeparator(" -> ") << endl;`
to print a nicely-formatted itinerary. You can similarly print the list of already-visited cities using the `toString()` or `toPrettyString()` methods.
- **WARNING:** Again, remember that `ArrayList` positions use 1-based indexing, not zero-based.

What to Turn In

Submit your answers to the questions from the first part of the lab in the file `lab6.txt` and your source code using the following command:

```
~csci132/bin/submit
```

Be sure you add your name and date to the prologue for each file you modify.