Algorithms & Data structures

Tutorial

L.J. Bel

7 March 2018

Introduction

This document presents a number of modules, each containing a coherent set of questions and exercises. Pick any module you deem interesting! If you don't like it, or it's too easy or too difficult, just switch to another. Each module has a difficulty rating, from 1 (easy) to 3 (difficult) stars. This difficulty is based on my estimate of how much experience one has with algorithms (and computer science in general). Each module also indicates the language used. The code can be found here: `github.com/lbel/icsc2018`.

To run the software you need at least the following:

Python Python 2.7, 3.4 or newer. You can find an installation of Python 2.7.5 on CentOS 7 at `lxplus7.cern.ch`, under `/usr/bin/python`.

C++ A C++ compiler supporting C++11, such as GCC 4.8 or newer. You can find an installation of GCC 4.8.5 on CentOS 7 at `lxplus7.cern.ch`, under `/usr/bin/g++`. This should be referenced automatically by the Makefiles.

Java JDK version 1.8 or newer. This is also installed on `lxplus7`, and will be picked up by the scripts.

Contents

1 ∗ Breadth-first and depth-first search

Breadth-first search (BFS) and depth-first search (DFS) are two rather fundamental concepts in computer science. This quick exercise will familiarise you with them! You'll also implement both a Stack and a Queue in the process.

DFS visits all nodes in a tree (or, in general, any graph structure) by starting at the head node, then moving along the first edge it sees. It keeps moving until it hits a node without children, after which it will backtrack to the last node it visited that still had children. We will implement this backtracking using a Stack. Once there are no more nodes with unvisited children, we are sure to have visited all the nodes! By itself this doesn't do anything yet, but this can be used to search for a node in the tree, or to apply an operation on each node of the tree.

Exercises

1. Go to the directory `trees` and open the file `dfs.py`.
2. Flesh out the Stack implementation, using the provided code (and a Python `list` as a backing data structure). What built-in methods can you use?
3. Now complete the DFS implementation.
   (a) Start by pushing the head of the tree (`tree.head`) onto the Stack.
   (b) Then loop until the Stack is empty, each time pushing the children of the current node onto the Stack.
   (c) Don't forget to add visited nodes to the `discovered` list!
4. Check your implementations by running the script. It will print two trees and the result of your iteration. Do you visit all the nodes, in the right order?

BFS is very similar to DFS, except it exhausts all the children of the current node before moving on to their children. Another way to see it is that it traverses a tree level-by-level (from left to right in the ASCII visualisation provided by the script).

Exercises

1. Open the file `bfs.py`.
2. Flesh out the Queue implementation, again using a Python `list` and as many built-in methods as you can find.
3. Complete the BFS implementation. Note that it's very similar to DFS!
4. Again, check your implementations by running the script.
5. Run the larger example (`example3`) by uncommenting the last line (and removing the one above it) in `tree.py`, both for DFS and BFS. Does it work?

2 $*$ Java collections (Java)

Java has very good out-of-the-box support for all the data structures discussed in the lectures. In this exercise, you will verify their complexity by running large numbers of operations on them. We'll be using the `java.util.Collection` interface, which is an interface common to all java collections. The advantage of this is that it supports similar operations through the same interface. For example, adding an object to a set and pushing an object to the back of an array both use the same method `add`.

Exercises

1. Look up the documentation of `java.util.Collection`. Try to understand what the different methods are supposed to be doing.
2. The code in `java_collections/JavaCollectionsTest.java` tests a collection by passing it (empty) to the method `runTest`. This method then reads a large number of strings from a text file, adds them to the collection, and performs various operations on it. Modify the code to pass it an empty `ArrayList` (an implementation of a dynamic array). This class is already imported in the file.
3. Run the code by executing `./run` in the `java_collections` directory. It should report the runtimes of various operations.
4. Run it again. What changed? Can you explain why?
5. Now run it with several different `Collection` instances, such as `LinkedList`, `TreeSet` and `HashSet`. Try to understand the differences in runtimes between them!
   - Note that you can add print statements in between the tests to make the output easier to understand. In Java, you can print using `System.out.println`.
6. Some of these classes have constructor parameters to tune their initial configuration. Look up which of them do, and try to pass sensible values to improve the runtimes.
   - If the runtime fluctuates too much to easily see the difference, try repeating the call to `readStrings` to increase the test sample size.

3 $*$ Recursion and stack overflow (Python)

Recursive implementations can simplify many algorithms, such as the implementation of the Fibonacci sequence given in `recursion/recursion.py`. This script takes n as an argument and outputs the n^{th} Fibonacci number. The recursion lies in the fact that the method `fib` repeatedly calls itself. This module is about inefficiencies and stack overflows (no, not the website).

Exercises

1. Run the example. Does it give the right answer?
2. Increase n. What do you observe?
3. What is the complexity of this implementation?
4. Of which concept, presented in the lectures, is this implementation an example?

This implementation is rather terrible. We can speed it up a lot by using a cache.

Exercises

1. Modify the code by adding a global cache and storing each answer in that cache after calculating.
2. If a cached answer is available, just use it instead of recalculating the result!
3. Make sure your code works up to at least $n = 500$.
4. Now try running with $n = 1000$. What happens?

When running code, the computer keeps track of all the called methods, so that it knows how to proceed once the current method returns. This is done in a buffer with a fixed size. That size is large enough that normally this causes no problems, but if we go very deep – as a recursive algorithm very well may – it is definitely possible to hit the limit. Let's fix the code!

Exercises

1. Replace the method `fib` with one that doesn't rely on recursion (ie., doesn't call itself).
2. Make sure it runs reasonably fast, even for $n > 1000$.
3. Yup, that's all – good luck!

You can find a cached implementation of the recursive algorithm in `recursion/recursion_cache.py`, and a solution of the final exercise in `recursion/recursion_solution.py`.

4 ∗∗ Bloom filters (Python)

A *Bloom filter* is an implementation of the `Set` ADT. For this exercise, assume this ADT supports the following operations:

`add(`x`)` Add item x to the `Set`. If x was already in the `Set`, return `false`; otherwise add it and return `true`.

`contains(`x`)` Check and return whether item x is currently in the `Set`.

Bloom filters do not support a `remove` operation, so you won't have to worry about that!

The Bloom filter is a stochastic data structure. That means there's a certain randomness to its operations. It functions not by storing the objects added to it, but rather by storing k different hashes of the objects in a bit array of size m. An object is added by calling each of the k hash functions on it, finding the bins in the bit array corresponding to those hashes , and setting each of those k bits to 1. To check whether an object is present, again each of the hash functions is called. If any of the bits is 0, the object is definitely not in the filter. If all the bits are 1, it probably is, but not necessarily – these bits may also have been flipped by a combination of other objects! Therefore, it is possible to get false positives with a Bloom filter, but never false negatives. The advantage of this data structure is its extremely high space efficiency as compared to other data structures, such as hash tables.

Questions

1. What is the space complexity of a Bloom filter?
2. What is the time complexity of a Bloom filter for the operations described above?

The file `bloom/bloom.py` contains a skeleton implementation of a Bloom filter class, as well as a small script to test it on words from `data/example_text.txt` and `data/random_strings.txt`. Make sure to run it from the `bloom` directory.

Exercises

1. Write the missing bodies for the `add` and `contains` methods. You may use the methods `__get_bit` and `__set_bit`. Note that `self.__hashes` is a list of functions!
2. Run the script. Do you get many false positives? And false negatives?
3. Increase m. Does this help reduce the number of false positives?
4. Add some hash functions to `self.__hashes`. What are good choices to prevent overlap with existing hash functions?

An example implementation with decent (but not perfect) performance can be found in `bloom/bloom_solution.py`.

5 ∗∗ Dijkstra's algorithm (Python)

In this module, you will implement Dijkstra's algorithm. A framework containing a `Graph` data structure and several test cases can be found in `dijkstra/dijkstra.py`. Everything below the method called `dijkstra` are test cases. The first is trivial, and the second is the one presented in the lectures.

Questions

1. Try to understand the structure of the three classes near the beginning of the file.
2. Look at the first two test cases, and see if you can map the nodes (vertices) and edges of the second test case to those in the lectures.

Next, let's implement the actual algorithm! A skeleton implementation has been set up in the method `dijkstra`.

Exercises

1. Fill out the rest of the method.
2. Run the file (from the directory `dijkstra`). Do both test cases give the correct result?
3. Uncomment and run the third test case. You can comment out the first two to suppress some output. These are actual roads of Rome! (Ok, in 1999. And the vertices are represented by arbitrary numbers, so don't bother trying to find your house.)
4. Critically analyse the complexity of your implementation. Are there any calls in there that could perform slowly, such as calls to functions of built-in data structures?
5. Try and run the last test case. This one consists of almost 20 MB of California road data – getting the complexity right is paramount for this one!
6. All edges in the last test case have equal weight (1). Can you use this knowledge to your advantage to speed up the runtime of your algorithm? Note that if you modify it under that assumption, the other test cases may not work anymore (unless you build a check into it; it's up to you if you want to do so).

6 $* * *$ Linked lists (C++)

Linked lists were presented in the lectures as an alternative to array-based sequential containers. This exercise explores some of their properties, strengths and weaknesses. The code relies heavily on C++14 features – don't hesitate to ask if you have any questions about that!

The folder `linked_list` contains two source files: `linked_list.h` features an almost compelete (singly-)linked list implementation, and `main.cxx` has some code to test it. There's also a small shell script to compile the code.

Exercises

1. Run `./compile` to compile the code, then try running the code with `./main`. You should see some arrays of numbers printed.
2. The program prints the time in ms spent to allocate elements to a `linked_list` and an `std::vector`. Which one is faster?
3. Look at `main.cxx`. You can pass it a number as an argument to insert a different number of elements (the default is 10). Run it again with a larger number of elements. At what point does the performance of the vector overtake that of the linked list? (Comment the call to `print_all` to see all the output.)

Now look at the linked list implementation. It features a lot of methods that coincide with the definitions of `std::vector`, including support for iteration (which is already used in the program to print the elements). However, there is a lot of duplicate and inefficient code.

Exercises

1. Copy `linked_list.h` to `linked_list_original.h` to preserve its original contents.
2. The current implementation of the `size` method is far less than optimal. Can you rewrite it so that it runs in $\mathcal{O}(1)$? You can (and should) change other parts of the class!
3. Add a reference to the last element of the linked list, and modify the `back` method to run in $\mathcal{O}(1)$ by using it. Do you need to update any of the modifier methods (`push_front`, `push_back`, `pop_front`, and `pop_back`) to keep the reference up-to-date.
4. Modify `push_back` to use the field as well, then do the same with `pop_back`.
5. How much faster does your code run compared to the original file? You can easily check by including the original file rather than the modified one in `main.cxx`. And how does your performance compare to that of `linked_list_solution.h`? And to the STL implementation (`linked_list_std.h`)?

Questions

1. What was the original complexity of the following methods?
2. And what is the compelxity of your new implementation?
   - `size`
   - `push_front`
   - `push_back`
   - `pop_front`
   - `pop_back`