Joshua Sloan
University of Wyoming
jsloan3@uwyo.edu

Homework 4
COSC 5110 - Analysis of Algorithms

1 Exercise 2.27

The square of a matrix A is its product with itself, AA.

(a)
Show that five multiplications are sufficient to compute the square of a 2 x 2 matrix.

Let M be an arbitrary 2 x 2 matrix.

We can label the 4 values of M separated into 4 quadrants, where
the top left value = A,
the top right value = B,
the bottom left value = C
and finally, the bottom right value = D, such that:

$$M = \begin{array}{|c|c|}\hline A & B \\ \hline C & D \\ \hline\end{array}$$

$$MM = \begin{array}{|c|c|}\hline A & B \\ \hline C & D \\ \hline\end{array} \text{ x } \begin{array}{|c|c|}\hline A & B \\ \hline C & D \\ \hline\end{array} = \begin{array}{|c|c|}\hline AA+BC & AB+BD \\ \hline CA+DC & CB+DD \\ \hline\end{array} = \begin{array}{|c|c|}\hline AA+BC & B(A+D) \\ \hline C(A+D) & CB+DD \\ \hline\end{array}$$

so we can see that the total number of multiplications required is 5 (1. AA, 2. BC (equal to CB given that multiplication is commutative), 3. B(A+D), 4. C(A+D) and finally 5. DD. □

(b)
What is wrong with the following algorithm for computing the square of an n x n matrix?

"Use a divide-and-conquer approach as in Strassen's algorithm, except that instead of getting 7 subproblems of size n/2, we now get 5 subproblems of size n/2 thanks to part (a). Using the same analysis as in Strassen's algorithm, we can conclude that the algorithm runs in time O($n^{log_2 5}$)."

We are only able to reduce the number of multiplications made when we are dealing with number multiplication, rather than matrix multiplication. Matrix multiplication is NOT commutative. In this regard, given two matrices A and B, AB $\neq$ BA in the same way that 2*3 = 3*2. We are only able to use this trick for a 2 x 2 matrix because every quadrant that we are using represents a single number value. For this reason, the aforementioned algorithm would not work. □

(c)
In fact, squaring matrices is no easier than matrix multiplication. Show that if n x n matrices can be squared in time O(n^c), then any two n x n matrices can be multiplied in time O(n^c).

Given: n x n matrices can be squared in O(n^c) time.
Let A and B be two n x n matrices that we want to multiply together.

Construct a new 2n x 2n matrix M, where the values in the top right quadrant are the values of the matrix A, the values in the bottom left quadrant are the values of the matrix B, and the top left and bottom right quadrants are zero matrices, such that:

$$M = \begin{array}{|c|c|}\hline 0 & A \\ \hline B & 0 \\ \hline\end{array}$$

Squaring the matrix M gives us $MM = \begin{array}{|c|c|}\hline 0 & A \\ \hline B & 0 \\ \hline\end{array} \text{ x } \begin{array}{|c|c|}\hline 0 & A \\ \hline B & 0 \\ \hline\end{array} = \begin{array}{|c|c|}\hline AB & 0 \\ \hline 0 & BA \\ \hline\end{array}$

Now, depending on whether we would like to multiply in the order A x B or B x A, we simply need to take the values from the respective quadrant to be our answer.

Because our matrix is 2n x 2n, the time to square M is O($2n^c$) = O(n^c),
$\therefore$ any two n x n matrices can be multiplied in O(n^c) time. □

2 Exercise 5.3

Design a linear-time algorithm for the following task.

Input: A connected, undirected graph G.
Question: Is there an edge you can remove from G while still leaving G connected?

Can you reduce the running time of your algorithm to O(|V|)?

edgeRemoval(G)

```
0: vertexEncountered = boolean array of size |V| (initialized to FALSE)
   while performing a breadth first search on G do
      if vertex V is newly visited then
         mark V as visited
      else if vertex V has already been marked then
         return TRUE
      end if
   end while
   return FALSE
```

If ever a vertex is encountered twice during the BFS, then we know that the second edge encountered is not needed in order for the graph to remain connected. For this reason, we can simply return TRUE, as we know that there is at least one edge that is not required for the graph to remain connected. No more than an additional linear amount of work (O(|V|)) would be required to create such a vertex encountering array, which does not change our algorithm's time-complexity.

Even in the case of a fully connected graph, the maximum number of edges needed to be analyzed before we encounter the first repeated vertex in our BFS would be equal to |V| (|V| - 1 edges for all other vertices in G from our starting vertex, plus 1 for the first edge check which causes the algorithm to return TRUE), making our worst case run-time of a TRUE response O(|V|).

In the worst case FALSE response, our algorithm must traverse all the way through the graph G before returning FALSE. In this regard, the run-time would be O(|V|*|E|). However, we know that if FALSE is returned, then there are no unnecessary edges in our graph, such that |E| = |V| - 1. In this regard we can simply upper bound O(|V|*|E|) $\leq$ O(2|V|) = O(|V|). □

3 Excercise 6.17

Given an unlimited supply of coins of denominations $x_1, x_2, ..., x_n$ we wish to make change for a value v, that is, we wish to find a set of coins whose total value is v. This might not be possible: for instance, if the denominations are 5 and 10 then we can make change for 15 but not for 12. Give an O(nv) dynamic-programming algorithm for the following problem.

Input: $x_1, ..., x_n$;v.
Output: Is it possible to make change for v using coins of denominations $x_1, ..., x_n$?

SEE NEXT PAGE

```
makeChange($x_1, ..., x_n$;v)
0: numSplits = zero array of size v + 1
0: numSplits[0] = 1
  for d = 1 to n do
    for i = 1 to v do
      if i < x$_d$ then
        do nothing //coin size is greater than the value we are trying to make change for
      else
        numSplits[i] = numSplits[i] + numSplits[(i - x$_d$)]
      end if
    end for
  end for
  if numSplits[v] $\geq$ 1 then
    return TRUE
  else
    return FALSE
  end if
```

This algorithm works by utilizing knowledge of a previous value's ability to have change made from specified denomination amounts. Starting with a zero array of size v+1, we initialize the first value to 1 (given that we can trivially make change for 0 using 0 coins). In this regard, our array values are indicative of the number of ways that we can make change for a specific amount. We know that if a number is divisible by the value of a particular coin denomination, then we are able to make change for that value. The value of the numSplits array, value(denomination) back from our current index, will accomplish this for us (building off the first instance where the index is equal to the value of the coin denomination).

Once we have gone through all coin denominations, the last index in our array will tell us whether we are able to make change using our coin denominations (if 0, then FALSE, otherwise TRUE).

It is important to note that this algorithm only works if coin denomination values are integer values and that our target value is represented in the same system (i.e. using the American monetary system, the target value is given in terms of cents, rather than a dollar/cents combination and all denominations are also given in terms of cents, such that five dollars would be denoted as 500).

The run-time of our algorithm is the linear construction of our array of size v+1 (O(v)), plus the time of the two nested for-loops from 1 to n and 1 to v (O(nv)) and finally, plus a constant-time check at the last index of our array to return either TRUE or FALSE (O(1)).

$\therefore$ the total run time of this algorithm is O(v) + O(nv) + O(1) = O(nv). $\square$

4

Recall the CLIQUE decision problem: given a graph G and a number k, does G have a clique of size k?

(a)

Show that if CLIQUE is in P, then there is a polynomial-time algorithm that takes a graph G as input and outputs the size of a largest clique in G.

Given that CLIQUE $\in$ P, we know that there is a polynomial-time algorithm for the CLIQUE decision problem.

To find the largest size clique in G, we simply need to run our polynomial-time decision problem on G, starting with k = |V| (i.e. the number of vertices in G). If there exists a clique of size k, return k. Otherwise, decrement k by 1 and run the polynomial time decision problem on G again with the new k value. If k reaches 1, return 1 (unless G is empty, in which case we can start with a preliminary check prior to the loop to return the base case of 0).

Because the size of our graph is bounded by a constant integer of vertices, the total running time for this algorithm is O(|V|*P$_t$), where P$_t$ is the polynomial run-time of our clique decision problem. Given that we compute the maximum clique using linear number of polynomial-time computations, it is clear that the total run-time $\in$ P.

∴ if CLIQUE is in P, then there is a polynomial-time algorithm that takes a graph G as input and outputs the size of a largest clique in G. □

(b)
Show that if CLIQUE is in P, then there is a polynomial-time algorithm that takes a graph G as input and outputs a largest clique in G.

First, run the algorithm from part (a) to determine the maximum clique size, k, for G.

Using this value of k, for every vertex in G we will remove the vertex (and any connected edges to that vertex) from G and check to see if there still exists a clique of size k using the polynomial-time clique decision algorithm. If TRUE, then we keep the revisions made to graph G, otherwise, append the vertex (and connected edges) back on to G and move on to the next vertex in G.

Once all vertices have been checked, return G.

Because this algorithm will remove all vertices and edges not contained in a maximum size clique, we know that all remaining vertices and edges must belong to a maximum size clique. It is important to note that this algorithm will only return 1 instance of the largest clique in G if our original graph contains more than 1 clique of size k.

Similarly to the algorithm from part (a), our algorithm runs a polynomial-time algorithm + a constant amount of work, a linear number of times ($|V|$). Consequently, the additional part of our algorithm also $\in$ P. The total run time for this algorithm is $O(P_a) + O(P_b)$ where P_a is the polynomial run-time of the algorithm from (a) and P_b is the polynomial run-time of our additional work, making the total run-time of this algorithm polynomial.

∴ if CLIQUE is in P, then there is a polynomial-time algorithm that takes a graph G as input and outputs a largest clique in G. □

I worked with Mason and Danny on this assignment.