Triangles In Homework 1, we gave an $O(nm)$ time algorithm to determine whether a graph has a triangle. This could be up to $O(n^3)$ time for dense graphs. Give an algorithm for the same problem running in time $O(n^p)$ for some $p < 3$. (Almost all points on the correctness proof.)

Look at the adjacency matrix $M_G$ for the graph. Let $A = (M_G)^2$ be the product of $M_G$ with itself. I claim that $A[i,j] > 0$ if and only if there is a node $k$ so that both $\{i, k\}$ and $\{j, k\}$ are edges.

By definition of matrix product, $A[i,j] = \sum_k M[i,k]M[k,j]$. If no $k$ as above exists, all the terms in this sum are 0, so $A[i,j] = 0$. If such a $k$ exists, the term corresponding to that $k$ is 1, and the other terms are non-negative, so $A[i,j] > 0$.

So the algorithm that suggests itself is: Compute $A$ using fast matrix multiply, e.g., Strassen’s $O(n^{\log_2 7})$ time algorithm or subsequent improvements. Then for each $i, j$, if there is an edge from $i$ to $j$, check if $A[i,j] > 0$. If so, there is a $k$ so that all three are connected, so return True. If we try all possibilities without finding such a pair, return False.

If there is a triangle $i, j, k$ in $G$, when we reach $i, j$, we will return True. If we return True on step $i, j$, there is a $k$ so that all three are connected, so the above algorithm returns True if and only if $G$ contains a triangle.

The total time is the time to perform the matrix multiply plus $O(n^2)$ for searching for all $i, j$. Thus the total time is the same as that for matrix multiply.

Base Conversion Give an algorithm that inputs an array of $n$ base binary digits representing a positive integer and outputs an array of ternary digits representing the same integer in base 3. Note that we are counting time in terms of single digit operations, so grade school operations like integer addition and multiplication do not take constant time. (For example, the FFT-based integer multiplication algorithm took $O(n \log^2 n)$ operations if we use the grade school method to multiply the complex numbers to within $O(\log n)$ bits of precision.) [6 points correct algorithm and correctness proof, 14 points efficiency]
The following recursive algorithm uses the divide and conquer method to convert an n bit binary integer \(x_{n-1}\ldots x_0\) into decimal. It uses the \(O(n^{\log_2 3})\) time divide-and-conquer multiplication algorithm Multiply2 from class and the text; and the grade school linear time \(O(n)\) Add algorithm as sub-routines. We assume Add and Multiply are defined to take decimal integers as input and output. Note that \(2^n\), in binary, is a 1 followed by \(n\) 0's, so is easy to construct as a binary integer in linear time. Let ConstructPower2, given \(n\), construct \(2^n\) in binary in time \(O(n)\).

ConvertToBinary\( (x_{n-1}\ldots x_0)\): Binary integer represented as an array of bits): decimal integer;

1. IF \(n = 1\) return \(x_0\).
2. \(y \leftarrow x_{n-1}\ldots x_{n/2}\)
3. \(z \leftarrow x_{n/2-1}\ldots x_0\)
4. \(w \leftarrow \text{ConstructPower2}(n/2 - 1)\) (in binary)
5. \(a \leftarrow \text{ConvertToBinary}(y)\)
6. \(b \leftarrow \text{ConvertToBinary}(z)\)
7. \(c \leftarrow \text{ConvertToBinary}(w)\)
8. \(c \leftarrow \text{Add}(c, c)\)
9. \(d \leftarrow \text{Multiply}(a, c)\)
10. \(e \leftarrow \text{Add}(d, b)\)
11. Return \(e\)

For the time analysis, we make three recursive calls, on \(w, y\) and \(z\). Now, \(w, y, z\) are all \(n/2\) bit binary integers. The time to construct them is \(O(n)\). The results are decimal versions, and so have fewer digits (by about a \(\log 10\) factor). Thus \(a, b, c, d\) are at most \(O(n)\) digits each, so the time for the two Adds is \(O(n)\) and the Multiply is \(O(n^{\log_3 2})\). SO the total time out of the recursion is \(O(n^{\log_3 2})\).

This gives \(T(n) = 3T(n/2) + O(n^{\log_3 2})\) as the recurrence. This meets the format of the Master Theorem with \(A = 3, B = 2, K = \log 3\). Then since \(3 = 2^{\log_3 3}\), we are in the steady-state case, so \(T(n) \in O(n^{\log_3 2} \cdot \log n)\).

Note that if we replace the multiplication procedure in the above algorithm by a faster one, e.g., the \(O(n \log^3 n)\) algorithm based on FFT, we still only save the \(\log n\) factor. However, we can get a big improvement if we observe that for \(n = 2^k\), all the powers of 2 we use in the divide and conquer are actually for \(2^i\), \(0 \leq i \leq \log n\). So we could pre-compute all of these using \(\text{Exponent}[i] = \text{Multiply(Exponent}(i - 1), \text{Exponent}(i - 1)]\) and then replace the recursive call to get \(c\) by a look-up in the array \(\text{Exponent}\). Constructing this table is a recursive procedure, assuming \(n\)
is a power of 2. Construct the table for $n/2$, then compute the entry for $n$ by a single multiplication. Since $2^n$ has $\Theta(n)$ decimal digits, this would be given by a recursion of the form $T(n) = T(n/2) + O(n \log^3 n)$, which is a top-heavy recurrence, giving total time $T(n) = O(n \log^3 n)$. Then the main procedure has two recursive calls, a look-up into the array, a multiplication, and an addition. Thus, the recurrence is $T(n) = 2T(n/2) + O(n \log^4 n)$, which is a roughly steady-state recurrence, giving $T(n) = O(n \log^4 n)$. Since the time to construct the table is smaller, this is also the order of the total time of the whole algorithm.

**Hamiltonian path** Consider the following algorithm for deciding whether a graph has a Hamiltonian Path from $x$ to $y$, i.e., a simple path in the graph from $x$ to $y$ going through all the nodes in $G$ exactly once. ($N(x)$ is the set of neighbors of $x$, i.e. nodes directly connected to $x$ in $G$).

1. $HamPath(G, x : node, y : node)$
2. If $x = y$ is the only node in $G$ return True.
3. If no node in $G$ is connected to $x$, return False.
4. For each $z \in N(x)$ do:
   5. If $HamPath(G - \{x\}, z, y)$, return true.
6. Return False

We want to find a Hamiltonian path from $x$ to $y$. The back-tracking algorithm branches on the first node visited after $x$. The choices are all the neighbors of $x$ in the graph. We can’t return to $x$ because the path must be simple. If we can find a Hamiltonian path from some neighbor $z$ of $x$ to $y$ in $G - \{x\}$, we can append $x$ to the front of the path and get a Hamiltonian path from $x$ to $y$. On the other hand, if we cannot for any neighbor $z$ of $x$, then we cannot find a Hamiltonian path from $x$ to $y$, since the second node along such a path must be such a $z$.

The time of the algorithm is proportional to the number of recursive calls that get made, since each time through the loop makes one recursive call and then does $O(1)$ work. The recursion tree for the program has depth $n$, since each time step we delete exactly one node from $G$. In general graphs, the time could be as much as $(n-1)!$ since the first node could have $n-1$ neighbors, each possibility for the second node $n-2$ neighbors, and so on. However, if no node in $G$ has more than 3 neighbors, an obvious upper bound for the total time is $O(3^n)$, since no node in the recursion tree has more than 3 children, and the depth is $n$.

In fact, since after the first step, we only call $y$ right after one of its neighbors $x$ has been deleted from the tree, we can see that the total size of the tree is at most $3 \times 2^{n-1} = O(2^n)$. 

3
There is a more subtle argument that improves this bound: A graph with \( n \) nodes none of which has more than 3 edges can have at most \( 3n/2 \) edges total, since each edge touches two nodes. Every time we branch in the recursion, i.e., \( x \) has more than one neighbor, we delete at least 2 edges from the graph. Thus, any path in the recursion tree can have at most \((3n/2)/2 = 3n/4\) places where there was any branching. Thus, the time is actually bounded by \( O(2^{75n}) \) for such graphs.

Finally, we can still improve this as follows. Let’s make a simple change to the above algorithm. If \( |E| < |V| - 1 \) then the graph is not connected, so it is certainly not Hamiltonian. So let’s return False if this is true.

Say that following a specific branch of the recursion, not counting the root, we branch in two ways \( t_2 \) times and in a single case \( t_1 \) times. Then the number of nodes remaining is \( n' = |V| - t_1 - t_2 - 1 \) and the number of edges remaining is at most \( m' = |E| - t_1 - 2t_2 \). \( m' - n' \leq |E| - |V| - t_2 + 1 \leq 1/2|V| - t_2 + 1 \) since \( |E| \leq 1.5|V| \) (as before). Thus, if \( t_2 > |V|/2 + 2 \), the branch is pruned. Thus, we get at most one three way branch and \( n/2 + 2 \) two way branches along any path, which bounds the number of leaves by \( 12 \times 2^{n/2} \). Since each path to a leaf is at most size \( n \), the total size of the tree is thus \( O(n^{2^{n/2}}) \), which is also a bound for the new algorithm’s running time.

**Big Bucks** Smallville is the last city on Earth not saturated by Big Bucks coffee shops. Smallville has one business street with \( n \) blocks. The profit associated with putting a coffee shop on block \( i \) in given in an array \( Profit \) as \( Profit[i] \). However, they cannot put coffee shops within \( d \geq 1 \) blocks from each other, i.e., if a shop is in block \( i \) then there cannot be one in blocks \( i - d, i - d + 1, i - 1 \) or \( i + 1, i + 2, \ldots i + d \).

An backtracking algorithm for computing the maximum total profit of BigBucks coffee shops is as follows:

\[
\text{BTBigBucks}(d, \text{Profit}[1..n])
\]

1. IF \( n = 0 \) return 0.
2. IF \( n \leq d + 1 \) return \( \max_{1 \leq i \leq n} \text{Profit}[i] \)
3. \( \text{Case}_1 \leftarrow \text{Profit}[1] + \text{BTBigBucks}(d, \text{Profit}[d+2..n]) \) {If we put a shop in block 1, we cannot put one in 2,..d+1}
4. \( \text{Case}_2 \leftarrow \text{BTBigBucks}(d, \text{Profit}[2..n]) \) {If we don’t put a shop in block 1, there are no other restrictions}
5. Return \( \max(\text{Case}_1, \text{Case}_2) \)

**Part 1: 5 points** Illustrate this algorithm on the following inputs: \( d = 2, n = 8, \text{Profit}[1..8] = 2, 4, 3, 7, 8, 4, 7, 5 \) (as a tree of recursive calls and answers).
We first branch on whether to build a store in the first block. If we do, we get 2 + a recursive solution to 7,8,4,7,5. If we don’t, we get a recursive solution to 4,3,7,8,4,7,5.

In the first case, we choose between 7 + BigBucks((7,5), 2) and BigBucks((8,4,7,5), 2). Since the first part has \( n = 2 \leq d = 2 \), we immediately return \( \max(7,5) = 7 \), which makes a total of 14. The second part, we choose between 8 + BigBucks((5), 2) = 8 + 5 = 13 and BigBucks(4,7,5) = \( \max(4 + \text{BigBucks}(\text{empty}, 2) = 4 + 0, \text{BigBucks}(7,5) = \max(7,5) = 7) = 7 \). Thus, the second case return 13. Since this is less than 14, the first case is 14.

In the second case, we choose between 4 + BigBucks((8,4,7,5), 2) and BigBucks((3,7,8,4,7,5), 2) We already simulated the algorithm on (8,4,7,5) above to get 13. The other option chooses between 3 + BigBucks((4,7,5), 2) and BigBucks((7,8,4,7,5), 2). We already simulated the algorithm on (4,7,5) and got 7, and on (7,8,4,7,5) and got 14. Thus, the max for the second option is 14, which is less than 4 + 13. So we would return 17 for the second case, which is greater than 14. So the final answer returned is 17, the profit from picking the second block, the fifth block and the last block.

**Part 2: 5 points** Give an upper bound on the number of recursive calls the above algorithm makes, in the worst-case. (Some points will be based on how tight the bound is. Be sure to explain your answer.)

The algorithm makes one recursive call to an input of size \( n - 1 \) and the other to an input of size \( n-d-1 \). Since \( d \geq 1 \), the second is of size at most \( n-2 \). Thus, \( T(n) \leq T(n-1) + T(n-2) + O(1) \), which grows at the same order as the \( n \)th Fibonacci numbers, \( O((1 + \sqrt{5}/2)^n) \) which is about \( 2.7^n \).

**Part 3: 10 points** Give a dynamic programming version of the recurrence.

The algorithm doesn’t change \( d \) in recursive calls, and only deletes initial segments of the array \( A \). So the recursive calls are all of the form \( \text{BigBucks}(A[1..n], d) \) for \( 1 \leq I \leq n \) (or the empty array, which we view as \( I = n + 1 \)). The matrix \( B[I] \) will compute and store the values \( \text{BigBucks}(A[I..n], d) \) using the same recurrence.

\[
\begin{align*}
\text{DPBigBucks}(A[1..n], d) \quad & 1. \text{ Initialize } B[1..n+1] \\
& 2. \quad B[n + 1] \leftarrow 0. \\
& 3. \text{ FOR } I = n \text{ downto } n - d \text{ do:} \\
& \quad 4. \quad B[I] \leftarrow \max(A[I], B[I + 1]) \\
& 5. \text{ FOR } I = n - d - 1 \text{ downto } 1 \text{ do:} \\
& \quad 6. \quad B[I] \leftarrow \max(A[I] + B[I + d + 1], B[I + 1])
\end{align*}
\]
7. Return B[1]

**Part 4: 5 points** Give a time analysis of this dynamic programming algorithm.

The algorithm has two loops, one of \( O(d) \leq O(n) \) and the other of \( O(n) \). So the total time is \( O(n) \).

**Part 5: 5 points** Show the array that your dynamic programming algorithm produces on the above example. \( d = 2, n = 8, A[1..8] = 2, 4, 3, 7, 8, 4, 7, 5 \)


**Implementation: Thresholds for Integer Multiplication** Implement the \( O(n \log^3 3) \) divide-and-conquer algorithm for integer multiplication from class, but with a threshold, below which naive “gradeschool” multiplication is used. Use an array of digits to represent inputs and outputs. Experimentally determine the optimal threshold. For what values of \( n \) do you see an improvement in the time using divide-and-conquer, both using no threshold and using the optimal threshold?

Many students confuse the optimal threshold with the break-even point.

Consider a bottom-heavy divide and conquer algorithm. For any fixed threshold \( T \), most of the work is being done at the small inputs, those less than \( T \).

If we use the break-even point as the threshold \( T \), then in fact, for inputs larger than \( T \), we should see no difference or minimal difference in the times with and without the threshold. We are replacing one algorithm on size \( T \) inputs with another that takes the same time.

If instead we use a value \( T \) where the grade-school method is a factor \( F \) over divide-and-conquer, we should see a factor \( F \) improvement on each recursive call to a size \( T \) sub-problem. Since very little of the total work is being done at larger sub-problems, we should see almost a factor \( F \) total improvement. The exact improvement will also depend on what fraction of work is being done on larger sub-problems, whether the divide-and-conquer algorithm makes recursive calls to size exactly \( T \) inputs, and how the programming language handles recursion. But typically, by using moderate sized thresholds, we can get a large constant factor improvement even for very large input sizes.

When describing an algorithm, don’t write out an entire pseudo-code; just describe it at a high level. Be sure to specify completely all data structures used
in the algorithm. Include correctness proofs and time analysis for all algorithms, except for the implementation problem.