For each of the algorithm problems, design as asymptotically efficient an algorithm as possible. Give a correctness argument (explanation, if it is relatively simple, or proof if not) and time analysis. You may use any well-known algorithm or data structure, or algorithm from the text or from class, as a sub-routine without needing to provide details.

**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)

**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 \(O(n \log^3 n)\) 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]

**Back-tracking** Give an improved exponential time algorithm for deciding whether a graph of degree at most 3 has a Hamiltonian path. (5 points correctness, 15 points efficiency and time analysis)

**Back-tracking and Dynamic Programming** Consider the following problem (a version of which was on the midterm).

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, we cannot put coffee shops within \(d \geq 1\) blocks from each other, i.e., if we put a shop in block \(i\) then we cannot put one in block \(i-d, i-d+1, i-1\) or \(i+1, i+2, \ldots i+d\).

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

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

1. If \(n = 0\) return 0.
2. If \(n \leq d + 1\) return \(\max_{1 \leq i \leq n} Profit[i]\)
3. Case1 \(\leftarrow Profit[1] + BT\text{BigBucks}(d, Profit[d+2..n])\) {If we put a shop in block 1, we cannot put one in \(2,..d+1\)}
4. Case2 ← BTBigBucks(d, Profit[2..n]) {If we don’t put a shop in block 1, there are no other restrictions}

5. Return max(Case1, Case2)

**Part 1: 2 points** Illustrate this algorithm on the following inputs: d = 2, n = 8, Profit[1..8] = 2, 4, 3, 7, 8, 4, 7, 5 (as a tree of recursive calls and answers).

**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.)

**Part 3: 10 points** Give a dynamic programming version of the recurrence, with a time analysis.

**Part 5: 3 points** Show the array that your dynamic programming algorithm produces on the above example.

**Implementation: Thresholds for Integer Multiplication** Implement the $O(n \log^3 n)$ divide-and-conquer algorithm for integer multiplication from class, but with a threshold, below which naïve “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?

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.