1. (5 pts) Use the characteristic polynomial method to solve this recurrence exactly:

$$T(n) = 9T(n-1) - \frac{1322}{49}T(n-2) + \frac{1320}{49}T(n-3)$$

with $T(0) = 7$, $T(1) = 148/7$ and $T(2) = 3132/49$. Show via induction that your solution is correct.

2. (20 pts total) The recursive algorithm to calculate the nth Fibonacci number is

```
Fib(n) {
   if n < 2
      return n
   else
      return Fib(n-1) + Fib(n-2)
   end
}
```

The time and space requirements are given by $R(n) = R(n-1) + R(n-2) + c$, which has solution $R(n) = O(\phi^n)$. We can compute $Fib(n)$ *much* faster if we use dynamic programming. Dynamic programming is a kind of recursive strategy in which instead of simply dividing a problem of size n into two smaller problems whose solutions can be merged, we instead construct a solution of size n by merging smaller solutions, starting with the base cases. We'll demonstrate this idea in three steps.

(a) (10 pts) First, consider this version of Fibonacci, which uses "memoization" (or simply memorization) to store the intermediate results in an array $F[n]$.

```
MemFib(n) {
   if n < 2
      return n
   else
      if F[n] = undefined
         F[n] = MemFib(n-1) + MemFib(n-2)
      end
      return F[n]
   end
}
```

(If you don't like the undefined trick, we could wrap this function in a second one that first allocates an array of size n and initialize each entry to $-\infty$. Then, call `MemFib(n)` after replacing the inner conditional with $F[n] = -\infty$.)

i. If we include the cost of the recursive calls, what is the running time? Explain.

ii. If we count only the additions, what is the running time? Prove this result by induction.

(b) (5 pts) But, we can do it even faster, and using less space. How? By eliminating the recursion completely and building up directly to the final solution by filling the F array in order.

```
DynFib(n) {
   F[0] = 0
   F[1] = 1
   for i = 2 to n
      F[i] = F[i-1] + F[i-2]
   end
   return F[n]
}
```

Now, what is the running time? Justify your claim.

(c) (5 pts) In this case, we can do one better because we do not need to store all the intermediate results.

```
FasterFib(n) {
   a = 0
   b = 1
   for i = 2 to n
      c = a + b
      a = b
      b = c
   end
   return b
}
```

Now, what are (i) the time and (ii) space requirements? Justify your claims, and comment briefly on how these compare to the original recursive function.

3. (30 pts total) Suppose you are given an array $A[1 \ldots n]$ of numbers, which may be positive, negative, or zero.

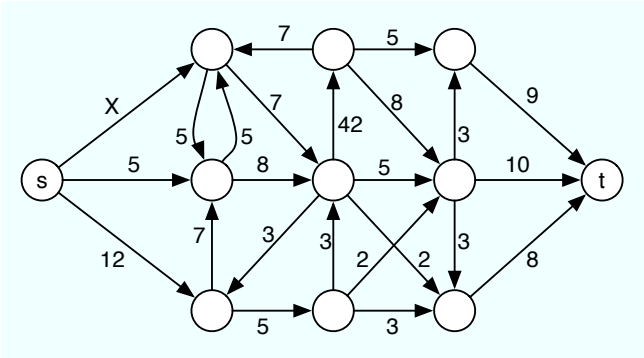
(a) (15 pts) Describe (explain in words and give pseudocode) and analyze (prove the correctness of and give the running time for) a dynamic programming algorithm that finds the largest sum of elements in a contiguous subarray $A[i \ldots j]$. For

example, if the input is the array $A = [-6, 12, -7, 0, 14, -7, 5]$, then the largest sum of any contiguous subarray is $19 = 12 - 7 + 0 + 14 = sum(A[2 \ldots 5])$.

Hint: Think about how you can compute, store and reuse intermediate results.

(b) (15 pts) Describe and analyze an algorithm that finds the largest product of elements in a contiguous subarray $A[i \ldots j]$. You may not assume that all the numbers in the input array are integers.

4. (10 pts total) The power company Watt Power is considering adding a new line connecting their source plant s to their distribution network G. However, the company engineers aren't sure how much additional power they will be able to push through G after adding the proposed line. The diagram below shows G', the network G plus the proposed line X.



(a) (5 pts) Make a diagram showing the minimum cut corresponding to the maximum flow for G. What is the weight of that cut? If Watt Power adds the connection X, what should be its capacity in order to maximize the increase in the flow across the network? Explain.

(b) (5 pts) Describe how to use the min-cut/max-flow algorithm to decide what capacity X should be used for an arbitrary graph $G = (V, E)$ and arbitrary proposed edge $(u, v) \notin E$ with capacity X.

5. (15 pts) Implement an efficient simulated annealing algorithm for the *Ising model*, a classic model of magnetization from statistical physics.[1] The Ising model is defined as a d-dimensional lattice S where each lattice element takes a value $s_i \in \{-1, +1\}$. The energy (score) of the system is given by

$$E = -\sum_{i \neq j} J_{ij} s_i s_j$$

where s_i, s_j are lattice sites and J_{ij} denotes the "coupling" strength between them. In the classic 2-dimensional *ferromagnetic* case, $J_{ij} > 0$ for all pairs of sites that are linked to each other in the lattice (the so-called von Neumann neighborhood) and $J_{ij} = 0$ for all other pairs. In the ferromagnetic case, the system energy is optimized (minimized) when all the lattice sites have the same value, either all -1 or all $+1$. Thus, this model can be viewed as modeling system-wide "frustration," with the global optima being defined, via E, as the states with no frustration across any pair of interacting elements.

For your implementation, use an $n \times n$ lattice ($d = 2$). Set $J_{ij} = 1$ for interacting sites (von Neumann neighborhood). Choose a geometric cooling schedule with α as a free parameter; use the Metropolis `accept()` function (from Lecture 13), adapted to prefer *decreases* in energy rather than increases; and, use the "single-flip" `neighbor()` function. This latter function chooses a uniformly random site in S and proposes to "flip" its state: if $s_i = -1$, then propose $s_i = +1$, and vice versa. Define convergence as n^2-in-a-row rejected proposals. Finally, use "periodic boundary conditions," in which a site $S_{1,1}$ has neighbors $S_{1,2}$, $S_{2,1}$, $S_{n,1}$ and $S_{1,n}$, i.e., "wrap" the left edge of the lattice around so that it neighbors the right edge and similarly for the top and bottom edges. (Topologically, this makes the lattice into a torus.)

Your implementation should take as input the initial configuration of the lattice $S^{(0)}$, an initial temperature t_0 and the annealing parameter α. When it terminates, it should output the final state $S^{(\text{stop})}$ and its corresponding energy $E_{S^{(\text{stop})}}$.

To make the annealing efficient, implement the `accept()` function so that it takes $\Theta(d)$ time rather than the naïve $\Theta(n^d)$ time to compute the energy E' of the proposed state.

[1] The Ising model exhibits a very rich range of dynamical behavior, including fun things like phase transitions and criticality. Its generalization to k states is called a Potts model, but basically works just the same.

6. (20 pts total) Use your implementation from 5 to conduct the following numerical experiments.

   (a) (15 pts) Let $n = 50$, $t_0 = 1000$, $\alpha = 1 - 10^{-4}$ and the initial state $S^{(0)}$ be one where each state s_i is chosen uniformly at random from $\{-1, 1\}$. Now, simulate the annealing process to find a minimum of the energy function.

   The deliverables here are (i) three snapshots of the system, one of the initial configuration, one when 80% of the sites have the same state and one showing the final annealed state, and (ii) a paragraph describing your results and what they mean with respect to local/global optima of E.

   Since each site has a binary value, you can render these snapshots as black-and-white bitmap images. To get the second image, you'll need to implement some measurement code to track the number of sites in either state and to take a snapshot when the system hits the target configuration. For each figure, give the temperature of the annealer at that moment, the energy of the configuration and the fraction of sites with the majority state.

   Note, it's unlikely that the final state will be the global minimum of E.

   (b) (5 pts) For the choices of n, t_0 and $S^{(0)}$ specified in (a), tinker with the annealing schedule (vary α) to find one that can consistently find the global optimum upon convergence. Describe the schedule, or explain why (in terms of the structure of the local optima) it is difficult to produce one that meets this criteria.

7. (*15 pts extra credit*) Currency arbitrage is a form of financial trading that uses discrepancies in foreign currency exchange rates to transform one unit of some currency into more than one unit of the same currency. For instance, suppose 1 U.S. dollar bought 0.82 Euro, 1 Euro bought 129.7 Japanese Yen, 1 Japanese Yen bought 12 Turkish Lira and one Turkish Lira bought 0.0008 U.S. dollars. Then, by converting currencies, a trader could start with 1 U.S. dollar and buy $0.82 \times 129.7 \times 12 \times 0.0008 \approx 1.02$ U.S. dollars, thus turning a 2% profit. Of course, this is not how real currency markets work because each transaction must pay a commission to a middle-man for making the deal.

   Suppose that we are given n currencies $c_1, c_2, \ldots, c_n$ and an $n \times n$ table R of exchange rates, such that one unit of currency c_i buys $R[i, j]$ units of currency c_j. A traditional arbitrage opportunity is thus a cycle in the induced graph such that the product of the edge weights is greater than unity. That is, a sequence of currencies $\langle c_{i_1}, c_{i_2}, \ldots, c_{i_k} \rangle$ such that $R[i_1, i_2] \times R[i_2, i_3] \times \cdots \times R[i_{k-1}, i_k] \times R[i_k, i_1] > 1$. Each transaction, however, must pay a commission, which is typically some α fraction of the transaction value, e.g., $\alpha = 0.01$ for a 1% rate.

(a) (*10 pts*) Give an efficient algorithm to determine whether or not there exists such an arbitrage opportunity, given a commission rate α. Analyze the running time of your algorithm.

Hint: It's possible to solve this problem in $O(n^3)$. Recall that Bellman-Ford can be used to detect negative-weight cycles in a graph.

(b) (*5 pts*) Explain what effect varying α has on the structure of the set of possible arbitrage opportunities your algorithm might identify.

8. (*25 pts total extra credit*) Suppose we want to maintain a dynamic set of values, subject to the following operations:

- `insert(x)`: Add x to the set (if it isn't already there)
- `printAndDeleteBetween(a,b)`: Print every element x in the range $a \leq x \leq b$, in increasing order, and delete those elements from the set.

For example, if our current set is $\{1, 5, 3, 4, 8\}$, then invoking `printAndDeleteBetween(4,6)` will print the numbers 4 and 5 and changes the set to $\{1, 3, 8\}$. For any underlying set, `printAndDeleteBetween(`$-\infty$`,`∞`)` prints its contents in increasing order and deletes everything.

(a) (*15 pts*) Describe and analyze a data structure that supports these operations, each with amortized cost $O(\log n)$, where n is the maximum number of elements in the set.

(b) (*5 pts*) What is the running time of your `insert` algorithm in the worst case?

(c) (*5 pts*) What is the running time of your `printAndDeleteBetween` algorithm in the worst case?