1. Short answers:

(a) Solve the recurrence $T(n) = 2T(n/3) + O(\sqrt{n})$.

Solution: $T(n) = O(n^{\log_3 2}) = O(n^{0.63093})$.

The ith level of the recursion tree sums to $2^i\sqrt{n/3^i} = \sqrt{n}\cdot(2/\sqrt{3})^i$. So the level sums form a ascending geometric series; only the number of leaves matters. The recursion tree has depth $\log_3 n$ and each level ℓ has 2^ℓ nodes, so the number of leaves is $2^{\log_3 n} = n^{\log_3 2}$. ■

Rubric: 1 point.

(b) Solve the recurrence $T(n) = 2T(n/7) + O(\sqrt{n})$.

Solution: $T(n) = O(\sqrt{n})$.

The ith level of the recursion tree sums to $2^i\sqrt{n/7^i} = \sqrt{n}\cdot(2/\sqrt{7})^i$. So the level sums form a descending geometric series; only value of the root matters. ■

Rubric: 1 point.

(c) Solve the recurrence $T(n) = 2T(n/4) + O(\sqrt{n})$.

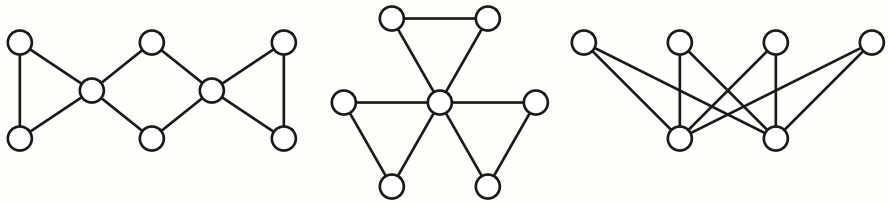
Solution: $T(n) = O(\sqrt{n}\log n)$.

Every level of the recursion tree sums to $\sqrt{n}$ and there are $\log_4 n$ levels. ■

Rubric: 1 point.

(d) Draw a connected undirected graph G with at most ten vertices, such that every vertex has degree at least 2, and no spanning tree of G is a path.

Solution: Here are three examples.

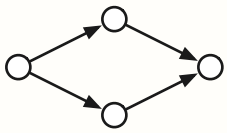


■

Rubric: 2 points. These are not the only correct solutions.

(e) Draw a directed acyclic graph with at most ten vertices, exactly one source, exactly one sink, and more than one topological order.

Solution: Here is the simplest example:



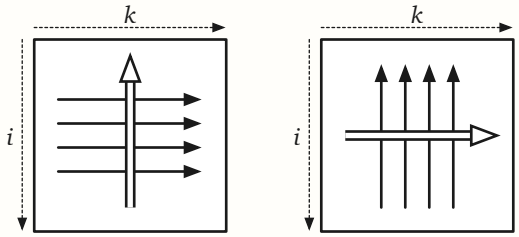
■

Rubric: 2 points. This is not the only correct solution.

(f) Describe an appropriate memoization structure and evaluation order for the following (meaningless) recurrence, and give the running time of the resulting iterative algorithm to compute $Pibby(1, n)$. (Assume all array accesses are legal.)

$$Pibby(i,k) = \begin{cases} 0 & \text{if } i > k \\ A[i] & \text{if } i = k \\ Pibby(i+1,k-1) + A[i] + A[k] & \text{if } A[i] = A[k] \\ \max \left\{ \begin{array}{c} Pibby(i+2,k), \\ Pibby(i+1,k-1), \\ Pibby(i,k-2) \end{array} \right\} & \text{otherwise} \end{cases}$$

Solution: $O(n^2)$ time. Here are two valid evaluation orders:



■

Solution: We can memoize *Pippy* into a two-dimensional array. We can fill this array in $O(n^2)$ time using two nested loops, one decreasing i and the other increasing k. (The nesting order doesn't matter.) ■

Rubric: 3 points = 1 for structure + 1 for evaluation order + 1 for time bound. These are not the only correct solutions.

2. Your company has two offices, one in San Francisco and the other in New York. Each week you decide whether you want to work in the San Francisco office or in the New York office. Depending on the week, your company makes more money by having you work at one office or the other. You are given a schedule of the profits you can earn at each office for the next n weeks. You'd obviously prefer to work each week in the location with higher profit, but there's a catch: Flying from one city to the other costs $1000. Your task is to design a travel schedule for the next n weeks that yields the maximum *total* profit, assuming you start in San Francisco.

   (a) ***Prove*** that the obvious greedy strategy (each week, fly to the city with more profit) does not always yield the maximum total profit.

Solution: Consider the following profit schedule:

SF	$1000	$0	$1000
NY	$0	$1000	$0

If we stay in San Fransisco all three weeks, we earn a profit of $2000, but if we follow the greedy strategy, our profit is only $1000. ■

Rubric: 2 points. This is not the only correct solution, but every correct solution requires an explicit counterexample.

(b) Describe and analyze an algorithm to compute the maximum total profit you can earn, assuming you start in San Francisco. The input to your algorithm is a pair of arrays $NY[1..n]$ and $SF[1..n]$, containing the profits in each city for each week.

Solution (dynamic programming): Let $MaxProfit(i, c)$ denote the maximum profit I can earn in weeks i through n, assuming that I am in city c just before week i begins. We need to compute $MaxProfit(1, SF)$. This function obeys the following recurrence:

$$MaxProfit(i,c) = \begin{cases} 0 & \text{if } i > n \\ \max\left\{\begin{array}{c} SF[i] + MaxProfit(i+1, SF) \\ NY[i] - 1000 + MaxProfit(i+1, NY) \end{array}\right\} & \text{if } c = SF \\ \max\left\{\begin{array}{c} SF[i] - 1000 + MaxProfit(i+1, SF) \\ NY[i] + MaxProfit(i+1, NY) \end{array}\right\} & \text{if } c = NY \end{cases}$$

We can memoize this function into an $n \times 2$ array $MaxProfit[1..n, 1..2]$ (using $1 = SF$ and $2 = NY$), which we can evaluate by decreasing i in the outer loop. The resulting algorithm runs in $\boldsymbol{O(n)}$ ***time***. ■

Rubric: 8 points, standard dynamic programming rubric (scaled). This is not the only correct dynamic programming solution.

Solution (dag traversal): We construct a directed acyclic graph $G = (V, E)$ as follows:

- $V = \{0, 1, 2, \ldots, n\} \times \{NY, SF\}$. Each node (i, c) denotes being in city c at the end of week i. ("The end of week 0" means before week 1.)
- E includes the following weighted edges for every index $i \geq 1$:
  - $(i-1, SF) \to (i, SF)$ with weight $SF[i]$
  - $(i-1, SF) \to (i, NY)$ with weight $NY[i] - 1000$
  - $(i-1, NY) \to (i, NY)$ with weight $NY[i]$
  - $(i-1, NY) \to (i, SF)$ with weight $SF[i] - 1000$

Altogether G has $2n$ vertices and $4n - 4$ edges. We need to compute the longest path from $(0, SF)$ to either (n, SF) or (n, NY). We can compute this longest path in $O(V + E) = \boldsymbol{O(n)}$ ***time*** using the dynamic programming / postorder traversal algorithm described in class. ■

Rubric: 8 points, standard graph reduction rubric (scaled). This is not the only correct graph-based solution.

3. Suppose you are given a directed graph $G = (V, E)$, whose vertices are either red, green, or blue. Edges in G do not have weights, and G is not necessarily a dag. The ***remoteness*** of a vertex v is the maximum of three shortest-path lengths:

   - The length of a shortest path to v from the closest red vertex
   - The length of a shortest path to v from the closest blue vertex
   - The length of a shortest path to v from the closest green vertex

   In particular, if v is not reachable from vertices of all three colors, then v is infinitely remote.

   Describe and analyze an algorithm to find a vertex of G whose remoteness is *smallest*.

Solution: Our algorithm proceeds as follows.

- Add three new vertices r, b, and g to the graph, with edges $r \to x$ for every red vertex x, edges $b \to y$ for every blue vertex y, and edges $g \to z$ for every green vertex z. The augmented graph has exactly $V + 3$ vertices and $E + V$ edges.
- Compute all shortest-path distances $dist(r, v)$ using breadth-first search from r.
- Compute all shortest-path distances $dist(b, v)$ using breadth-first search from b.
- Compute all shortest-path distances $dist(g, v)$ using breadth-first search from g.
- Compute the remoteness of every node v by brute force using the formula

$$remoteness(v) = \max\left\{dist(r, v),\ dist(b, v),\ dist(g, v)\right\} - 1$$

  (The -1 adjusts for the new edges that we added from r, b, and g.)
- Find a node v with minimum remoteness by brute force.

The overall algorithm runs in $\boldsymbol{O(V + E)}$ ***time***. ■

Rubric: 10 points, standard graph reduction rubric (brute force construction).

- -1 for invoking Dijkstra instead of BFS.
- -1 for *explaining* BFS instead of just writing "BFS" or "breadth-first search".
- Max 7 points for an algorithm that runs in $O(V^3)$ or $O(V^3 \log V)$ time when $E = \Theta(V^2)$.
- No penalty for implicitly assuming that $V = O(E)$ in the time analysis. However, any correct algorithm must return ∞ if (for example) the red vertices are completely disconnected from the rest of G.

This is not the only correct solution.

4. Suppose you are given an array $A[1..n]$ of integers such that $A[i]+A[i+1]$ is even for *exactly one* index i. In other words, the elements of A alternate between even and odd, except for exactly one adjacent pair that are either both even or both odd.

   Describe and analyze an efficient algorithm to find the unique index i such that $A[i]+A[i+1]$ is even. For example, given the following array as input, your algorithm should return the integer 6, because $A[6]+A[7] = 88+62$ is even. (Cells containing even integers are shaded blue.)

17	40	23	72	39	88	62	13	40	53	92	21	10	73	68

Solution: We can solve this problem in $O(\log n)$ ***time*** using the following variant of binary search.

```
ParitySwitch(A[1..n]):
  ℓ ← 1          ⟨⟨left index⟩⟩
  r ← n − 1      ⟨⟨right index⟩⟩

  while r − ℓ > 374
      m ← ⌊(ℓ + r)/2⌋
      if A[m] + A[m + 1] is even
          return m
      else if A[1] + A[m] + m is odd
          ⟨⟨m is too small⟩⟩
          ℓ ← m
      else ⟨⟨if A[1] + A[m] + m is even⟩⟩
          ⟨⟨m is too large⟩⟩
          r ← m
  use brute force
```

■

Rubric: 10 points = 1 point for "binary search" + 2 points for base case + 5 points for recursive cases + 2 points for running time. This is not the only correct formulation of this algorithm.

- −1 for each off-by-one error.
- −2 for each parity error (swapping even and odd).
- Max 3 points for a brute-force $O(n)$-time algorithm.

5. A *zigzag walk* in in a directed graph G is a sequence of vertices connected by edges in G, but the edges alternately point forward and backward along the sequence. Specifically, the first edge points forward, the second edge points backward, and so on. The *length* of a zigzag walk is the sum of the weights of its edges, both forward and backward.

Suppose you are given a directed graph G with non-negatively weighted edges, along with two vertices s and t. Describe and analyze an algorithm to find the shortest zigzag walk from s to t in G.

Solution: Let $G = (V, E)$ be the input graph, and let $w(u \to v)$ denote the weight of any edge $u \to v \in E$. First we construct a new directed graph $G' = (V', E')$ as follows:

- $V' = V \times \{0, 1\}$
- $E' = \big\{(u,0)\to(v,1), (v,1)\to(u,0) \;\big|\; u\to v \in E\big\}$
- Each edge $(u,0)\to(v,1)$ and $(v,1)\to(u,0)$ has weight $w(u\to v)$.

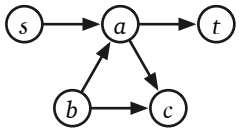
Our new graph G' has exactly $2V$ vertices and $2E$ edges. Vertex $(v, 0)$ means "I am at vertex v and my next edge must be forward," and vertex $(v, 1)$ means "I am at vertex v and my next edge must be backward." Every walk in G' corresponds to a zigzag walk in G'. Because G' is symmetric—the reversal of every edge is also an edge—we can also think of G' as an *undirected* graph.

Then we compute the shortest path from $(s, 0)$ to either $(t, 0)$ or $(t, 1)$ using Dijkstra's algorithm. The entire algorithm runs in $O(E' \log V') = \boldsymbol{O(E \log V)}$ ***time***. ■

Rubric: 10 points, standard graph reduction rubric (brute force construction).

- **No penalty for assuming the graph is unweighted and using BFS instead of Dijkstra (as long as the time analysis is consistent)**
- −1 for *explaining* Dijkstra's algorithm instead of just writing "Dijkstra's algorithm".
- Max 7 points for an algorithm that runs in $O(V^3)$ or $O(V^3 \log V)$ time when $E = \Theta(V^2)$.
- No penalty for implicitly assuming that $V = O(E)$ in the time analysis, or assuming that there is at least one zigzag walk from s to t.

This is not the only correct solution. However, any algorithm that considers only simple paths in G (which cannot repeat vertices or edges) cannot be correct. In the example below, every zigzag walk from s to t passes through vertex a at least twice; in particular, the shortest zigzag walk is $s\to a\leftarrow b\to c\leftarrow a\to t$.



On the other hand, it's not hard to prove that the *shortest* zigzag walk never traverses the same edge more than once, and in particular never traverses the same edge both forward and backward.