- Solutions worth full credit are shown in black.
- Alternate solutions and/or additional information is shown in blue.

1. (a) Suppose $A[1..n]$ is a sorted array of n distinct integers. Describe and analyze an efficient algorithm that either computes an index i such that $A[i] = i$ or correctly reports that no such index exists.

   (b) Now suppose every integer $A[i]$ is positive. Describe and analyze an even faster algorithm that either computes an index i such that $A[i] = i$ or correctly reports that no such index exists.

Solution:

- Consider a fictional array $B[1..n]$ where $B[i] = A[i] - i$ for all i. We can find an index i such that $B[i] = 0$ in $\boldsymbol{O(\log n)}$ ***time*** via binary search, because the array B is sorted in non-decreasing order. To avoid constructing B explicitly, just replace every instance of $B[i]$ in the binary search algorithm with $A[i] - i$.

```
FindIndex(A[1..n]):
  if n = 0
      return False
  m ← ⌊(n+1)/2⌋
  if A[m] = m
      return m
  else if A[m] < m
      return FindIndex(A[m+1..n])
  else ⟨⟨if A[m] > m⟩⟩
      return FindIndex(A[1..m-1])
```

```
FindIndex(A[1..n]):
  i ← 1; k ← n
  while i ≤ k
      j ← ⌊(i+k)/2⌋
      if A[j] = j
          return j
      else if A[j] < j
          i ← j+1
      else ⟨⟨if A[j] > j⟩⟩
          k ← j-1
  return False
```

- If $A[1] = 1$, return True, else return False. The algorithm runs in $\boldsymbol{O(1)}$ ***time***.

■

Rubric: Max 10 points:

(a) 7 points = 5 for algorithm (1 for base case + 2 for each recursive case) + 2 for time analysis. The algorithm can be described in English, recursive pseudocode, or iterative pseudocode; only one description is necessary for full credit. No penalty for off-by-one errors or missing floors/ceilings, unless they lead to infinite loops. A correct $O(n)$-time algorithm is worth at most 2 points. -1 if the best algorithm for part (a) is actually given in part (b).

(b) 3 points = 2 for algorithm + 1 for time analysis. A slower correct algorithm is worth at most 1 point, but only if it's faster than the given algorithm for part (a).

For example, a $O(n)$-time algorithm for part (a), together with a $O(\log n)$-time algorithm for part (b) that actually solves part (a), is worth at most 6 points.

A proof of correctness is *not* required for full credit; proofs are only required on exams if we explicitly ask for them (or if you submit a greedy algorithm).

2. ***Prove*** that it is NP-hard to determine whether a given graph contains a double-Hamiltonian circuit.

Solution: I'll describe a reduction from the standard Hamiltonian cycle problem.

Let G be an arbitrary graph. For each vertex v in G, add three new vertices v_1, v_2, v_3 and four new edges $vv_1, v_1v_2, v_1v_3, v_2v_3$. Call the resulting graph H.

($\Leftarrow$) Suppose G has a Hamiltonian cycle C. For each vertex v, replace the subpath $u\to v\to w$ in C with the walk $u\to v\to v_1\to v_2\to v_3\to v_2\to v_3\to v_1\to v\to w$. The resulting walk visits each of the vertices v, v_1, v_2, v_3 exactly twice, and is therefore a doubly-Hamiltonian circuit in H.

($\Rightarrow$) Suppose H has a double-Hamiltonian circuit D. For each original vertex v from G, the circuit D must visit both v and v_1, and therefore must traverse the edge vv_1 twice, once in each direction. Between those two traversals, D must visit each of the vertices v_1, v_2, v_3 exactly twice. Thus, if we remove the subwalk $v\to v_1\to\cdots\to v_1\to v$ from D, for every vertex v, the resulting walk is a Hamiltonian cycle in G.

Thus, G has a Hamiltonian cycle if and only if H has a double-Hamiltonian circuit.

Clearly, H can be constructed from G in polynomial time. ■

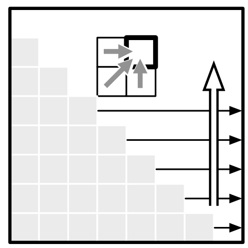
Rubric: Max 10 points: 3 for a correct reduction + 3 for 'if' proof + 3 for 'only if' proof + 1 for 'polynomial time'. For example, an incorrect polynomial-time reduction with a correct 'if' proof is worth at most 4 points.

3. Describe and analyze an algorithm to find the length of the longest subsequence of a given string that is also a palindrome.

Solution: Suppose the input string is $A[1..n]$. Let $MaxPal(i, j)$ denote the length of the longest palindrome subsequence of $A[i..j]$; we need to compute $MaxPal(1, n)$.

$$MaxPal(i, j) = \begin{cases} 0 & \text{if } i > j \\ 0 & \text{if } i = j \\ 2 + MaxPal(i+1, j-1) & \text{if } i < j \text{ and } A[i] = A[j] \\ \max\{MaxPal(i+1, j), MaxPal(i, j-1)\} & \text{otherwise} \end{cases}$$

We can memoize this function into a 2d array $MaxPal[1..n+1, 0..n]$. We can fill this array row by row from the bottom up, filling each row from left to right, in $\boldsymbol{O(n^2)}$ ***time***.



```
MaxPalSubseq(A[1..n]):
  for i ← n+1 down to 1
      for j ← i−1 to n
          if i > j
              MaxPal[i, j] ← 0
          else if i = j
              MaxPal[i, j] ← 1
          else if A[i] = A[j]
              MaxPal[i, j] ← 2 + MaxPal[i+1, j−1]
          else
              MaxPal[i, j] ← max {MaxPal[i+1, j], MaxPal[i, j−1]}
  return MaxPal[1, n]
```

■

Solution: The longest palindrome subsequence in a string A is a longest common subsequence between A and its reversal. We can compute the reversal of A in $O(n)$ time, and then compute the longest common subsequence in $\boldsymbol{O(n^2)}$ ***time*** using the algorithm Jeff described in class last week. ■

Rubric: Max 10 points, standard dynamic programming rubric; see the HW2 solutions. This is not the only correct traversal order for the first solution. An $O(n^3)$-time algorithm is worth at most 7 points; an $O(n^4)$-time algorithm is worth at most 4 points; any slower algorithm is worth at most 3 points; scale partial credit. Yes, the second solution really is worth full credit.

A proof of correctness is *not* required for full credit; proofs are only required on exams if we explicitly ask for them (or if you submit a greedy algorithm).

4. Suppose you are given a magic black box that can determine **in polynomial time**, given an arbitrary graph G, the number of vertices in the largest complete subgraph of G. Describe and analyze a **polynomial-time** algorithm that computes, given an arbitrary graph G, a complete subgraph of G of maximum size, using this magic black box as a subroutine.

Solution: Suppose the input graph has V vertices and E edges. The following algorithm calls the magic box $O(V)$ times, and otherwise runs in $O(V+E)$ time. (Deleting a vertex requires also deleting its incident edges, and each edge is considered for deletion at most twice.)

```
BuildMaxClique(G):
  k ← MagicBox(G)
  for all vertices v
      if MagicBox(G − v) = k
          G ← G − v
  return G
```

■

Rubric: Max 10 points: 7 points for algorithm + 3 points for time analysis. A time bound of $O(n^2)$, where n is the number of vertices, is worth full credit. This is not the only correct solution.

A proof of correctness is *not* required for full credit; proofs are only required on exams if we explicitly ask for them (or if you submit a greedy algorithm).

5. Describe and analyze an algorithm that computes the maximum sum of marked cells from a $4 \times n$ grid, subject to the condition that marked cells cannot be adjacent.

Solution: Let $A[1..n, 1..4]$ denote the input grid.

Call a subset of $\{1,2,3,4\}$ is *legal* if it does not contain two consecutive integers. There are exactly eight legal subsets: $\varnothing$, $\{1\}$, $\{2\}$, $\{3\}$, $\{4\}$, $\{1,3\}$, $\{1,4\}$, and $\{2,4\}$.

For any integer i and any subset $X \subseteq \{1,2,3,4\}$, let $MaxSum(i,X)$ denote the maximum possible score from the first i rows of the grid, assuming X indicates which cells in row $i+1$ are already marked. We need to compute $MaxSum(n,\varnothing)$.

$$MaxSum(i,X) = \begin{cases} -\infty & \text{if } X \text{ is not legal} \\ 0 & \text{if } i = 0 \\ \max\left\{ MaxSum(i-1,Y) + \sum_{j \in Y} A[i,j] \;\middle|\; Y \cap X = \varnothing \right\} & \text{otherwise} \end{cases}$$

This function can be memoized into an array $MaxSum[0..n, 0..15]$, by treating any subset of $\{1,2,3,4\}$ as a four-bit integer. We can fill the array in standard row-major order in ***$O(n)$ time***. (The big-Oh notation hides a constant of $8^2 \cdot 4 = 256$, but so what?)

```
MaxSum(A[1..n, 1..4]):
  for i ← 0 to n
      for all legal subsets X ⊆ {1,2,3,4}
          if i = 0
              MaxSum[i,X] ← 0
          else
              MaxSum[i,X] ← −∞
              for all legal subsets Y ⊆ {1,2,3,4} \ X
                  useY ← MaxSum[i−1,Y]
                  for all j ∈ Y
                      useY ← useY + A[i,j]
                  MaxSum[i,X] ← max{MaxSum[i,X], useY}
  return MaxSum[n,∅]
```

■

Rubric: Max 10 points; standard dynamic programming rubric. This is not the only correct solution.

A proof of correctness is *not* required for full credit; proofs are only required on exams if we explicitly ask for them (or if you submit a greedy algorithm).

(scratch paper)

(scratch paper)

(scratch paper)