1. The binomial coefficients are defined by the recurrence equation:

\[
\binom{n}{k} = \begin{cases} 
1, & \text{if } k = 0 \text{ or } k = n, \\
\binom{n-1}{k-1} + \binom{n-1}{k}, & \text{if } 0 < k < n.
\end{cases}
\]

Design an algorithm that computes the value of the binomial coefficient \(\binom{n}{k}\) in time \(O(nk)\).

Solution:

**Algorithm 1:** Binomial coefficients via Pascal’s triangle.

```plaintext
1 function BINOM (n,k); 
   Input: Integers n and k with 0 \leq k \leq n.
   Output: The binomial coefficient \(\binom{n}{k}\).
2 Declare auxiliary array \(b[i,j]\) for \(i = 0,\ldots,n\) and \(j = 0,1,\ldots,k\).
3 for i ← 0 to n do
4     for j ← 0 to k do
5         Set \(b[i,j]\) ← 0
6     end
7     Set \(b[i,0]\) ← 1
8     if \(i \leq k\) then
9         Set \(b[i,i]\) ← 1
10    end
11 end
12 for i ← 1 to n do
13     for j ← 1 to k do
14         Set \(b[i,j]\) ← \(b[i-1,j-1]\) + \(b[i-1,j]\)
15     end
16 end
17 return \(b[n,k]\)
```
2. Analyze the worst-case time complexity of the following algorithm:

**Algorithm 2:** Binary search.

```plaintext
1 function BINARYSEARCH (A[1...n], a);
   Input: Integer array A[1...n] with elements in increasing order, integer a.
   Output: Index k such that A[k] = a, or fail if no such k exists.
2   Set l ← 1, r ← n;
3   if l > r then return fail;
4   Set k ← ⌊(l + r)/2⌋
5   if A[k] < a then
6       Set l ← k + 1, go to line 3
7   if A[k] > a then
8       Set r ← k − 1, go to line 3
9   return k.
```

**Solution:** Denote by $M$ the number of times line 3 is executed. It is immediate from the structure of the algorithm that the running time is $\Theta(M)$.

It remains to derive bounds for $M$. We claim that when line 3 is executed for the $j$th time, with $j = 1, 2, \ldots, M$, the values of the variables $l$ and $r$ satisfy

$$r_{j+1} - l_{j+1} + 1 \leq 21^{1-j+\log n}.$$

Let us proceed by induction on $j$. For $j = 1$ the claim is immediate. So let us assume that the claim holds for $j$ and consider the case $j + 1$. Denote by $r_j$ and $l_j$ the values of the variables $r$ and $l$ when line 3 is executed for the $j$th time.

If line 6 gets executed before the $(j+1)$th execution of line 3, we have

$$r_{j+1} - l_{j+1} + 1 = r_j - (k + 1) + 1$$
$$= r_j - \lfloor (l_j + r_j)/2 \rfloor$$
$$\leq r_j - (l_j + r_j - 1)/2$$
$$\leq (r_j - l_j + 1)/2$$
$$= 21^{1-j+\log n}/2$$
$$= 21^{1-(j+1)+\log n}.$$

If line 8 gets executed before the $(j+1)$th execution of line 3, we have

$$r_{j+1} - l_{j+1} + 1 = k - 1 - l_j + 1$$
$$= \lfloor (l_j + r_j)/2 \rfloor - l_j$$
$$\leq (l_j + r_j)/2 - l_j$$
$$\leq (r_j - l_j + 1)/2$$
$$= 21^{1-j+\log n}/2$$
$$= 21^{1-(j+1)+\log n}.$$

This completes the induction.

It is immediate from the structure of the algorithm that each execution of line 3 sees $r - l$ decrease by at least one from the previous execution of line 3. Thus, when $r - l + 1 \leq 1$ the algorithm terminates in at most two executions of line 3. By the claim, $r_j - l_j + 1 \leq 1$ holds for all $j \geq 1 + \log n$. Thus, $M \leq 3 + \log n$ and hence $M = O(\log n)$. 

3. [Dasgupta et al., Ex. 2.5] Solve, to an exact closed-form expression, each of the following recurrences. You may assume in each case that \( n \) has an algebraically convenient form to ease the solution. Thus, you may assume in (a) that \( n = 3^k \) for \( k = 1, 2, \ldots \) and in (c) that \( n = 2^{2^k} \) for \( k = 0, 1, 2, \ldots \). The base case is \( T(1) = 1 \) in (a) and (b), and \( T(2) = 1 \) in (f).

(a) \( T(n) = 2T(n/3) + 1 \)

**Solution:** Let \( n = 3^k \). Expanding the recurrence, we observe that
\[
T(3^k) = 2T(3^{k-1}) + 1
\]
\[
= 2(2T(3^{k-2}) + 1) + 1
\]
\[
= 2(2(2T(3^{k-3}) + 1) + 1) + 1
\]
\[
= 2^k T(3^0) + \sum_{i=0}^{k-1} 2^i
\]
\[
= 2^k (1) + \sum_{i=0}^{k-1} 2^i
\]
\[
= \sum_{i=0}^{k-1} 2^i
\]
\[
= 2^k - 1.
\]

Thus, \( T(n) = 2^{\log_3 n} - 1 \). Equivalently, \( T(n) = 2n^\log_3 2 - 1 \).

(b) \( T(n) = 2T(n - 1) + 1 \)

**Solution:** Expanding the recurrence, we have
\[
T(n) = 2T(n - 1) + 1
\]
\[
= 4T(n - 2) + 2 + 1
\]
\[
= 8T(n - 3) + 4 + 2 + 1
\]
\[
= \sum_{k=1}^{n} 2^{k-1}
\]
\[
= \sum_{k=0}^{n-1} 2^k.
\]

Thus, \( T(n) = (1 - 2^n)/(1 - 2) = 2^n - 1 \).

(c) \( T(n) = T(\sqrt{n}) + 1 \)

**Solution:** Let \( n = 2^{2^k} \). Thus, \( \sqrt{n} = n^{1/2} = 2^{2^{k-1}} \). Expanding the recurrence, we have
\[
T(2^{2^k}) = T(2^{2^{k-1}}) + 1
\]
\[
= T(2^{2^{k-2}}) + 1 + 1
\]
\[
= T(2^{2^{k-3}}) + 1 + 1 + 1
\]
\[
= T(2^{2^0}) + k
\]
\[
= T(2) + k
\]
\[
= 1 + k.
\]

Thus, \( T(n) = 1 + \log \log n \) and hence \( T(n) = O(\log \log n) \).
4. [Dasgupta et al., Ex. 2.14] You are given an array of \( n \) elements, and you notice that some of the elements are duplicates; that is, they appear more than once in the array. Show how to remove all duplicates from the array in time \( O(n \log n) \).

**Solution:** Assuming that the array elements are comparable, we can sort the array, for example, using merge sort in time \( O(n \log n) \). Suppose the sorted array is \( A[1], A[2], \ldots, A[n] \) with \( A[1] \leq A[2] \leq \cdots \leq A[n] \). Then, for each \( i = 1, 2, \ldots, n-1 \), output \( A[i] \) if and only if \( A[i] < A[i+1] \). Finally, output \( A[n] \).

**Caveat:** Suppose we have access only to a black box (subroutine) that given \( i = 1, 2, \ldots, n \) and \( j = 1, 2, \ldots, n \) as input, returns only whether the elements at positions \( i \) and \( j \) are identical (true) or not (false). Then we require at least \( n(n-1)/2 = \Omega(n^2) \) calls in the worst case to the subroutine to remove all duplicates.

Consider an arbitrary algorithm \( A \) that correctly deletes duplicates on all inputs. Without loss of generality we can assume that \( A \) returns only a list of array positions that must be deleted to get rid of duplicates; that is, \( A \) does not manipulate the input array.

To reach a contradiction, suppose that \( A \) makes at most \( n(n-1)/2 - 1 \) calls to the subroutine when given as input an array with no duplicates. (That is, the subroutine always returns false.) Because \( A \) makes at most \( n(n-1)/2 - 1 \) calls to the subroutine, there is at least one input \((i_0, j_0)\) with \( i_0 \neq j_0 \) such that \( A \) does not call the subroutine with \((i_0, j_0)\). Consider now what happens if we run \( A \) on an array where the elements at \( i_0 \) and \( j_0 \) are identical, and the other \( n-2 \) elements are distinct. In particular, with the exception of the input \((i_0, j_0)\), the subroutine gives identical return values for this array and the array with all elements distinct. Thus, algorithm \( A \) cannot tell apart these two arrays because it does not call the subroutine with input \((i_0, j_0)\). In particular, algorithm \( A \) will output an empty list of duplicates even if there is a duplicate in the array where the elements at \( i_0 \) and \( j_0 \) are identical. This is a contradiction because \( A \) was assumed to be correct on all inputs. Thus, there is an input that requires at least \( n(n-1)/2 \) calls to the subroutine.
 Dasgupta et al., 2.20] You are given an array $x[1...n]$ of integers with $0 \leq x_i \leq 2^b - 1$ for $i = 1, 2, \ldots, n$. Assume that the array elements can be copied and moved in time $O(1)$. Design an algorithm that sorts the array in time $O(bn)$, and carefully justify the bound. Why doesn’t the $\Omega(n \log n)$ lower bound on sorting apply here?

**Solution:** Let us write $x_i$ in base 2 as $x_i = \sum_{j=0}^{b-1} 2^{b-1-j}x_{ij}$, with $x_{ij} \in \{0, 1\}$ for all $j = 0, 1, \ldots, b-1$. Note that we have $x_{i1} < x_{i2}$ if and only if there exists a $k = 0, 1, \ldots, b-1$ such that $x_{ik} < x_{i\bar{k}}$ and for all $j = 0, 1, \ldots, k-1$ we have $x_{ij} = x_{i\bar{j}}$. Equivalently, $x_{i1} < x_{i2}$ holds if and only if, viewing $x_{i1}$ and $x_{i2}$ as $b$-bit strings, the first difference between $x_{i1}$ and $x_{i2}$ occurs in the $k$th-most significant bit, which is a 0 in $x_{i1}$ and a 1 in $x_{i2}$.

We can turn this observation into a divide-and-conquer sorting algorithm as follows. Suppose our task is to sort the subarray $x[\ell...h]$, and we are promised that the elements of $x[\ell...h]$ all agree pairwise in their most significant $k-1$ bits. (Initially, take $k = 1$, $\ell = 1$, $h = n$. In particular, the promise is trivial.)

Let us carry out the sort as follows. First, split the array $x[\ell...h]$ so that no element with the $k$th bit set to 1 is to the right of an element with the $k$th bit set to 0. This splitting can be implemented in $O(h - \ell)$ time. Indeed, scan $x[\ell...h]$ from the left towards the right until an element with the $k$th bit set to 1 is found, and from the right towards the left until an element with the $k$th bit set to 0 is found, or until the two scans meet. Whenever such a pair of elements is found, transpose the elements and continue the scan. When the two scans meet, the split is complete. After the split, the left part of the array consists of elements that agree in the most significant $k$ bits, with the $k$th bit set to 0, and the right part of the array consists of elements that agree in the most significant $k$ bits, with the $k$th bit set to 1. We can thus sort the left and right parts using two recursive calls. (Indeed, the promise holds for $k+1$ in the left and right parts.) Note, however, that one of the parts may be empty, in which case only one recursive call is required. The base case is at $k = b+1$, at which point we observe that the array $x[\ell...h]$ consists of identical elements and hence is sorted.

There are $b+1$ levels of recursive calls, and the time required at each level $k$ is bounded by the total time consumed by the split operations at that level, which is $O(n)$ since the parts at each level are pairwise disjoint. Thus, the running time is $O(bn)$.

The $\Omega(n \log n)$ lower bound on sorting does not apply here because we are using the structure of the encoding of integers with at most $b$ bits. Recall that the lower bound holds if the sorting process is based exclusively on pairwise comparison between elements.
6. [Dasgupta et al., Ex. 1.8] Justify the correctness of the recursive division algorithm (Textbook page 26; Lecture 3 slide 12) and show that it takes time $O(n^2)$ on $n$-bit inputs.

**Solution:** The division algorithm is shown below. The result $(q, r)$ is correct if $x = qy + r$ and $0 \leq r \leq y - 1$. This is clearly the case when $x = 0$. Let us proceed by induction in the number of bits $m$ in $x$, that is, $m \geq 1$ and $2^{m-1} \leq x < 2^m$.

On line 6, $\text{divide}(\lfloor \frac{x}{2} \rfloor, y)$ is computed correctly by assumption, because $\lfloor \frac{x}{2} \rfloor$ has $m - 1$ bits. Thus, after line 6, $\lfloor \frac{x}{2} \rfloor = qy + r$ and $0 \leq r \leq y - 1$. After line 7, we have $2 \lfloor \frac{x}{2} \rfloor = qy + r$ and $0 \leq r \leq 2y - 2$. If $x$ is even, $2 \lfloor \frac{x}{2} \rfloor$ is just $x$. Otherwise, $2 \lfloor \frac{x}{2} \rfloor = x - 1$. In either case, the computation on lines 8–11 gives us $x = qy + r$ and $0 \leq r \leq 2y - 1$. Lines 12–15 maintain the value of the expression $qy + r$ while reducing $r$ to the range $0 \leq r \leq y - 1$. Therefore, on line 16, we have $x = qy + r$ and $0 \leq r \leq y - 1$, thus the return value is correct.

Let $n$ denote the maximum of the bit widths of $x$ and $y$. The algorithm calls itself with recursion depth $m = O(n)$, and performs at each recursion level a constant number of $O(n)$-time operations (doubling, addition, subtraction, comparison). Therefore, the algorithm terminates in time $O(n)O(n) = O(n^2)$.

---

**Algorithm 3: Division**

```
function divide(x, y);
    Input: Two $n$-bit integers $x$ and $y$, where $y \geq 1$
    Output: Their quotient and remainder of $x$ divided by $y$

    if $x = 0$ then
        return $(q, r) = (0, 0)$;
    else
        $(q, r) = \text{divide}(\lfloor \frac{x}{2} \rfloor, y)$;
        $q = 2 \times q$, $r = 2 \times r$;
        if $x$ is odd then
            $r = r + 1$;
        end
        if $r \geq y$ then
            $r = r - y$, $q = q + 1$;
        end
        return $(q, r)$;
    end
```
7. [Dasgupta et al., Ex. 5.21] A feedback edge set of an undirected graph \( G = (V, E) \) is a subset of edges \( E' \subseteq E \) that intersects every cycle of the graph. Thus, removing the edges \( E' \) will render the graph acyclic.

Give an efficient algorithm for the following problem:

Input: Undirected graph \( G = (V, E) \) with positive edge weights \( w_e \).
Output: A feedback edge set \( E' \subseteq E \) of minimum total weight \( \sum_{e \in E'} w_e \).

Solution:
Let us prove the following property of a minimum \( \text{fbes} \) (feedback edge set, abbreviation defined for this solution only):

Let \( G \) be an undirected graph. \( E' \subseteq E \) is a minimum weight \( \text{fbes} \) of \( G \) iff \( E' = E \setminus \{E_1, E_2, ..., E_k\} \) is maximum weight spanning forest of \( G \).

By a spanning forest of a graph \( G \) with connected components \( G_1, G_2, \ldots, G_k \) we mean a sub-forest consisting of \( k \) trees, each spanning a different component \( G_i \).

Proof. Assume \( E_1 \) is a min-weight \( \text{fbes} \). This implies directly that \( E_1 \) is acyclic. For the sake of contradiction, assume \( E_1 \) is not connecting all vertices in some \( G_i \). Then one can move an edge \( f \) from \( E_1 \) to \( E_1 \) such that no cycles appear in \( E_1 \). Thus \( E_1 \setminus \{f\} \) is still intersecting each cycle while it has smaller weight than \( E_1 \), contradicting minimality of \( E_1 \). We conclude \( E_1 \) must be connecting each \( G_i \) — thus it must be a spanning forest.

Note that we did not yet argue why \( E_1 \) is actually a max-weight spanning forest. However, let us continue from assuming another set \( E_2 \subseteq E \) is a max-weight spanning forest. It has no cycles so \( E_2 \) must be a \( \text{fbes} \).

Let us calculate some weights now. Denote the weight of a set \( X \subseteq E \) by \( W_X = \sum_{e \in X} w_e \) and notice the property \( W_X = W_E - W_X \). The max-weight property \( W_{E_2} \geq W_{E_1} \) implies \( W_{E_2} \leq W_{E_1} \). But \( E_1 \) was a min-weight \( \text{fbes} \), implying \( W_{E_2} = W_{E_1} \) and \( E_2 \) is a min-weight \( \text{fbes} \) as well.

Similarly we get \( W_{E_2} = W_{E_1} \) and \( E_1 \) is max-weight spanning tree.

From the above, we can extract both parts of the standard \( \text{lhs-implies-rhs and rhs-implies-lhs} \) argument, and the proof is complete.

We remark that Kruskal’s min-weight spanning tree algorithm actually obtains a min-weight spanning forest for graphs that are not connected. A simple max-weight spanning forest algorithm is obtained by reversing the choice between large and small weights in Kruskal’s algorithm. The complement is the asked minimum \( \text{fbes} \).

A pseudocode is shown below. The union-find operations are run in the graph \( (V, X) \).
Algorithm 4: Algorithm for finding min-weight feedback edge set in undirected graphs

function fbe-set(G);

Input: Undirected graph \( G = (V, E) \) with weights \( w_e > 0 \ \forall e \in E \)

Output: Minimum weight feedback edge set \( E' \subseteq E \)

for each \( u \in V \):
    makeset\((u)\);

\( X = \{ \} \);

\( E' = \{ \} \);

Sort the edges \( E \) by weight;

for all edges \( \{u,v\} \in E \), in decreasing order of weight:
    if \( \text{find}(u) \neq \text{find}(v) \):
        add edge \( \{u,v\} \) to \( X \);
        union\((u,v)\);
    end
    else
        add edge \( \{u,v\} \) to \( E' \);
    end

return \( E' \);
8. [Dasgupta et al., Ex. 6.8] Given two strings \( x = x_1 x_2 \cdots x_m \) and \( y = y_1 y_2 \cdots y_n \), we wish to find the length of their longest common substring, that is, the largest \( k \) for which there are indices \( i \) and \( j \) with \( x_{i+k-1} \cdots x_{i+1} = y_{j+k-1} \cdots y_{j+1} \). Show how to do this in time \( O(mn) \).

Solution:

It is not easy to find an efficiently computable recursion for the longest common substring directly. Let us instead define \( L(i, j) \) as the length of the common suffix in \( X_i = x_1 x_2 \cdots x_i \) and \( Y_j = y_1 y_2 \cdots y_j \), that is, \( L(i, j) \) is the maximum value of \( k \) such that the \( X_i \) and \( Y_j \) string ends match: \( x_{i-k+1} x_{i-k+2} \cdots x_i = y_{j-k+1} y_{j-k+2} \cdots y_j \). Now \( L \) follows the simple recursion

\[
L(i, j) = \begin{cases} 
1 + L(i-1, j-1) & \text{if } x_i = y_j \\
0 & \text{if } x_i \neq y_j
\end{cases}
\]

The boundary values are \( L(0, j) = 0, L(i, 0) = 0 \).

The longest common substring of \( x \) and \( y \) must be a common suffix of some \( X_i, Y_j \). Thus the length of the longest common substring equals \( S(m, n) = \max_{1 \leq i \leq m} \max_{1 \leq j \leq n} L(i, j) \), and it is enough to find the values of \( L(i, j) \).

These observations give rise to the following algorithm:

\begin{verbatim}
Algorithm 5: Algorithm for longest common substring problem

function longest-common-substring(x, y);
    Input: Strings x, y
    Output: Size of longest common substring of x, y
    m ← length(x);
    n ← length(y);
    L[·, ·] ← zeromatrix[0...m, 0...n];
    max ← 0;
    for 1 ≤ i ≤ m do
        for 1 ≤ j ≤ n do
            if x[i] = y[j] then
                L[i, j] ← L[i-1, j-1] + 1;
                if L[i, j] > max then
                    max ← L[i, j];
            end
        end
    end
    return max;
\end{verbatim}
9. [Dasgupta et al., Ex. 6.2] You are going on a long trip. You start on the road at mile post 0. Along the way there are \( n \) hotels, at mile posts \( a_1 < a_2 < \cdots < a_n \), where each \( a_i \) is measured from the starting point. The only places you are allowed to stop are at these hotels, but you can choose which of the hotels you stop at. You must stop at the final hotel (at distance \( a_n \)), which is your destination.

You would ideally like to travel 200 miles a day, but this may not be possible (depending on the spacing of the hotels). If you travel \( x \) miles during a day, the penalty for that day is \((200 - x)^2\). You want to plan your trip so as to minimize the total penalty—that is, the sum, over all travel days, of the daily penalties.

Give an efficient algorithm that determines the optimal sequence of hotels at which to stop.

Solution:

Let \( P(k) \) be the minimum penalty incurred by a sequence of stops that starts at mile post 0 and ends at mile post \( a_k \).

Set \( P(0) = 0 \) and \( a_0 = 0 \). For \( 1 \leq k \leq n \), we have

\[
P(k) = \min\{ (200 - (a_k - a_j))^2 + P(j) : 0 \leq j \leq k - 1 \}.
\]

Indeed, consider an optimal sequence \( S_k \) of stops that starts at mile post 0 and ends at mile post \( a_k \). Let \( a_j \) be the second last stop in \( S_k \), \( 0 \leq j \leq k - 1 \). We claim that the subsequence \( S_j \) of \( S_k \) that starts at mile post 0 and ends at mile post \( a_j \) has to be optimal among sequences of stops that start at mile post 0 and end at \( a_j \). Indeed, if \( S_j \) is not optimal, we have a better sequence \( S'_j \) that starts at 0 and ends at \( a_j \), after which we could drive from \( a_j \) to \( a_k \) with strictly less total penalty than in \( S_k \); this contradicts the optimality of \( S_k \).

An \( O(n^2) \) dynamic programming algorithm for computing \( P(n) \) is immediate from the recurrence for \( P(k) \). A corresponding sequence of hotels is obtained by keeping track for each \( k \) of a value of \( j \) for which \( P(k) = (200 - (a_k - a_j))^2 + P(j) \). Let \( p[k] \) be such a value. Then, \( p[n], p[p[n]], p[p[p[n]]], \ldots, 0 \) is an optimal sequence of hotels.
10. Give a dynamic programming algorithm for the *shortest reliable path* problem:

**Input:** A graph $G$ with nonnegative lengths on the edges, two vertices $s$ and $t$ of $G$, and a nonnegative integer $k$.

**Output:** A shortest path from $s$ to $t$ that uses at most $k$ edges, or assert that no such path exists.

**Solution:**

Since all the edge lengths $\ell(e)$ are nonnegative, to every walk of length $L$ from $u$ to $v$ with at most $j$ edges there is a path of length at most $L$ from $u$ to $v$ with at most $j$ edges. (Indeed, any closed subwalk starting and ending at a repeated vertex has nonnegative length and hence can be deleted until no more repeated vertices exist, that is, a path is obtained.)

For vertices $u, v$ and a nonnegative integer $j$, let $D(u, v, j)$ be the shortest-path distance from $u$ to $v$, taken over paths with at most $j$ edges.

For $j = 0$, the only possible path is a path with no edges, and thus

$$D(u, v, 0) = \begin{cases} 
0 & \text{if } u = v; \\
\infty & \text{if } u \neq v.
\end{cases}$$

For $j \geq 1$, a shortest path $P$ from $u$ to $v$ with at most $j$ edges. Such a path $P$ consists of at most $j - 1$ edges or exactly $j$ edges. In the latter case there is a last edge $(w, v)$ in $P$ and the first $j - 1$ edges form a path from $u$ to $w$. This path is a shortest path from $u$ to $w$ with at most $j - 1$ edges. (Proof: Suppose not and let $Q$ be a shorter path from $u$ to $w$ with at most $j - 1$ edges. Then the walk $W$ formed by taking $Q$ followed by $(w, v)$ contradicts the choice of $P$ if $W$ is a path. So suppose $W$ is not a path. Then, the only possible repeated vertex is $v$. Since edge lengths are nonnegative, we can delete the closed subwalk to obtain a path with at most the same length as $W$, which again contradicts the choice of $P$.) We can thus find the distance by taking the minimum of the cases:

$$D(u, v, j) = \min\{D(u, v, j - 1) \cup \{D(u, w, j - 1) + \ell(w, v) : (w, v) \in E\}\} .$$

Algorithm 6 gives a dynamic programming algorithm with running time $O(n^3)$, where $n$ is the number of vertices. (A faster algorithm could be obtained by modifying Dijkstra’s algorithm to keep track of the number of edges traversed.)
Algorithm 6: Shortest $k$-hop path via dynamic programming

function shortest-k-hop-path($G, \ell, s, t, k$);

Input: Graph $G = (V, E)$ with $V = \{1, 2, \ldots, n\}$, lengths $\ell : E \to \mathbb{Z}_{\geq 0}$, vertices $s, t \in V$, nonnegative integer $k$

Output: Distance $D[s, t, k]$ and predecessor vertices $t, p[r], p[p[r]], \ldots, s$ on a shortest path from $s$ to $t$ with at most $k$ edges

1. for $1 \leq u \leq n$ do
2. $p[u] \leftarrow \perp$;
3. for $1 \leq v \leq n$ do
4. $D[u, v, 0] \leftarrow \infty$;
5. if $i = j$ then
6. $D[u, v, 0] \leftarrow 0$
7. end
8. end
9. for $1 \leq j \leq k$ do
10. for $1 \leq v \leq n$ do
11. $D[s, v, j] \leftarrow D[s, v, j - 1]$;
12. for $(w, v) \in E$ do
13. if $D[s, v, j] > D[s, w, j - 1] + \ell(w, v)$ then
14. $D[s, v, j] \leftarrow D[s, w, j - 1] + \ell(w, v)$;
15. $p[v] \leftarrow w$
16. end
17. end
18. end
19. end
20. end
21. end

Caveat. If there were edges with negative length, we would have no guarantee that subpaths of shortest paths are shortest paths. In this case we can nevertheless still use a dynamic programming recurrence that keeps track of the set of vertices $S \subseteq V$ spanned by the current path.

Let $D(S, u, v)$ be the shortest-path distance from $u$ to $v$, taken along paths that visit each of the vertices in $S \subseteq V$ exactly once, start at $u$, and end at $v$.

For all subsets $S \subseteq V$ with $|S| = 1$, we have

$$D(S, u, v) = \begin{cases} 0 & S = \{u\} = \{v\}; \\ \infty & \text{otherwise}. \end{cases}$$

A shortest path with $|S| \geq 2$ vertices has to have a second last vertex. Thus, for $|S| \geq 2$,

$$D(S, u, v) = \min \{D(S \setminus \{v\}, u, w) + \ell(w, v) : (w, v) \in E \}.$$

This recurrence can be translated to a dynamic programming algorithm with running time $O(n^{k+1})$. 

11. [Dasgupta et al., Ex. 8.4] Given a graph $G$ and a nonnegative integer $k$ as input, the CLIQUE problem asks us to decide (yes/no) whether there exists a set $C$ of $k$ vertices in $G$ such that any two vertices in $C$ are joined by an edge.

(a) Show that CLIQUE is an NP-type problem in the sense defined in Lecture 10, slide 14.

Solution:
Given $G$, $k$, and a candidate solution $C$ as input, it is straightforward to check that $C$ has size $k$, and (by iterating over all pairs of vertices $u, v \in C, u \neq v$) that any two vertices of $G$ are joined by an edge. Assuming that $G$ has $n$ vertices, this takes $O(n^2)$ time if $G$ is given as an adjacency matrix, or $O(n^3)$ if $G$ is given as an adjacency list. In both cases the checking algorithm runs in time polynomial in the instance size, which is $\Theta(n^2)$.

(b) Consider now the CLIQUE-3 problem which is the CLIQUE problem restricted to graphs where every vertex has degree at most 3. What is wrong with the following proof of NP-completeness for CLIQUE-3?

We know that the CLIQUE problem in general graphs is NP-complete, so it is enough to present a reduction from CLIQUE-3 to CLIQUE. Given a graph $G$ with vertices of degree $\leq 3$, and a parameter $k$, the reduction leaves the graph and the parameter unchanged: clearly the output of the reduction is a possible input for the CLIQUE problem. Furthermore, the answer to both problems is identical. This proves the correctness of the reduction and, therefore the NP-completeness of CLIQUE-3.

Solution:
The reduction goes the wrong way. To show that a problem $P$ is NP-complete, we must first show that the problem is a search problem, and then present a polynomial-time reduction from an NP-complete problem $C$ to the problem $P$. 

12. [Dasgupta et al., Ex. 2.33] Suppose you are given three real matrices $A, B, C$ of size $n \times n$ and you are to decide (yes/no) whether $AB = C$. You can do this in $O(n^{\log_2 7})$ steps using Strassen’s algorithm. In this exercise we will explore a much faster, $O(n^2)$ time Monte Carlo test.

(a) Let $v \in \{0, 1\}^n$ be an $n \times 1$ vector selected uniformly at random from $\{0, 1\}^n$. Show that if $M$ is an $n \times n$ real matrix with at least one nonzero entry, then $P(Mv = 0) \leq 1/2$, where the probability is over the random choices of $v$.

**Solution:**
Let $c$ be a column where $M$ has some nonzero values, and let $v' = v_0v_1 \ldots v_{c-1}\overline{v}_cv_{c+1} \ldots v_{n-1}$, i.e. $v$ where only bit $v_c$ has been flipped. Now, at least one of $Mv$ and $Mv'$ must be non-zero, because $Mv - Mv' = M(v - v')$ is non-zero; this is because $(v - v')$ only has one non-zero bit, at position $c$, and $M$ has nonzero values in column $c$. Hence, for a vector $v$ chosen uniformly at random, $Mv = 0$ with probability at most $1/2$.

(b) Show that $P(ABv = Cv) \leq 1/2$ if $AB \neq C$.

**Solution:**
This is an immediate consequence of the previous result, with $M = AB - C$.

(c) Give pseudocode for an $O(n^2)$ time Monte Carlo algorithm for checking whether $AB = C$.

**Solution:**
In order to get error probability $1/2^k$, repeat the following test $k$ times: toss a coin $n$ times to get a vector $v$, and then compute $A(Bv) - (Cv)$. If $(AB - C)v \neq 0$, return “$AB \neq C$”, else proceed to the next repeat.

This algorithm takes $O(kn^2)$ steps: note that both $Bv$ and $Cv$ are vectors, each computable in $O(n^2)$ steps. Therefore, $A(Bv)$ can also be computed in time $O(n^2)$, and finally computing $A(Bv) - (Cv)$ takes $O(n)$ steps.
13. The Fibonacci numbers \( F_0, F_1, \ldots \) are given by the recurrence \( F_{n+1} = F_n + F_{n-1}, F_0 = 0, F_1 = 1 \).

(a) Show that for any \( n \geq 1 \), \( \gcd(F_{n+1}, F_n) = 1 \).

**Solution:** Let us show \( \gcd(F_{n+1}, F_n) = 1 \) by induction in \( n \).

**Base step** \( n = 1 \). Clearly \( \gcd(F_2, F_1) = \gcd(1, 1) = 1 \).

**Inductive step** \( n > 1 \). Assume as the inductive hypothesis \( \gcd(F_n, F_{n-1}) = 1 \). Let \( d = \gcd(F_{n+1}, F_n) \). As \( d \) divides \( F_{n+1} \) and \( F_n \), it also divides \( F_{n+1} - F_n = F_{n-1} \). As \( d \) is a common divisor of \( F_n \) and \( F_{n-1} \), it is also a divisor of \( \gcd(F_n, F_{n-1}) = 1 \), hence \( d = 1 \).

(b) Show that Euclid’s algorithm takes \( n - 1 \) iterations to compute \( \gcd(F_{n+1}, F_n) \).

**Solution:** The claim holds for \( n \geq 2 \). Given the pair \( (F_{n+1}, F_n) \), we observe that Euclid’s algorithm (shown below) follows the Fibonacci sequence down to \( (F_3, F_2) \) and then terminates after one more iteration.

**Algorithm 7:** Euclid’s algorithm for finding the greatest common divisor of two nonnegative integers

\[
\begin{align*}
\text{function} & \quad \text{Euclid}(a, b); \\
\text{Input:} & \quad \text{Two nonnegative integers } a \text{ and } b \\
\text{Output:} & \quad \gcd(a, b) \\
\text{if} & \quad b = 0 \quad \text{then} \\
\quad & \quad \text{return } a; \\
\text{else} & \quad \text{return } \text{Euclid}(b, a \mod b); \\
\end{align*}
\]

Let us proceed by induction.

**Base step** \( n = 2 \). With the input \( (F_3, F_2) = (2, 1) \), Euclid’s algorithm iterates once, calling Euclid(1, 0) which returns immediately.

**Inductive step** \( n > 2 \). Assume that it takes \( n - 2 \) iterations to compute \( \gcd(F_n, F_{n-1}) \). Given \( (F_{n+1}, F_n) \), the algorithm calls itself with the input \( (F_n, F_{n+1} \mod F_n) = (F_n, F_{n-1}) \), which totals to \( n - 2 + 1 = n - 1 \) iterations.

(c) Using Tutorial Problem 1:7, justify the claim that “the Fibonacci numbers are the worst case for Euclid’s algorithm”.

**Solution:** With any \( n \)-bit inputs, Euclid’s algorithm terminates within \( 2n \) iterations (Lecture 13, slide 32). Therefore, the number of iterations cannot be more than linear in \( n \). But is it possible to reach a worst-case bound lower than \( O(n) \)? The answer is no, and the Fibonacci sequence shows us why.

According to Tutorial Problem 1:7, there is a constant \( c < 1 \) such that \( F_k \leq 2^{ck} \) for all \( k \geq 0 \). That is, the number of bits in \( F_k \) is at most \( k \). For any \( k \geq 2 \), there is an input pair consisting of at most \( k + 1 \)-bit numbers, namely \( (F_{k+1}, F_k) \), on which Euclid’s algorithm performs \( k - 1 \) iterations, as shown above. If \( n \) denotes the number of bits in \( F_{k+1} \), then \( n \leq k + 1 \), from where it follows that the number \( k - 1 \) of iterations is at least \( n - 2 \). On the other hand, the number of iterations is at most \( 2n \). Therefore, given two subsequent \( n \)-bit Fibonacci numbers, Euclid’s algorithm performs \( \Theta(n) \) iterations, which exhibits the asymptotical worst-case behavior of the algorithm.