1.

(a)

Algorithm 1: Computing det(A) by Leibniz's method

```
Input: Matrix A [a_ij] of n × n
Output: Determinants of matrix A
Global: B                    /*A variable storing full permutation of 1, 2, 3, ··· , n*/
Function det(A):
    for i ← 1 to n do
        b[i] ← i;
    end
    full_perm (b[n],1);
    sum ← 0;                                  /*The determinant of matrix A*/
    for i ← 1 to n! do
        c ← B[i];
        count_rev ← 0;                        /*The counter of reverse order in c*/
        prod ← 1;                             /*One term in Lebniz's expansion*/
        for j ← 1 to n do
            prod ← prod·a_{j,c[j]};
            for k ← j+1 to n do
                if c[j] > c[k] then
                    count_rev ← count_rev+1;
                end
            end
        end
        sum ← sum +(−1)^{count_rev}·prod;
    end
    return sum;
end
Function full_perm (b[n], start):
    /*Description: Given a sequence b[n] return the full permutation of the
      subsequence from "start" to index n. All permutation are stored in some
      global variable B, and we can use some index to get them. */
    if start = n then
        for i ← 1 to n do
            c[i]=b[i];
        end
        store c to B;
    end
    for i ← start to n do
        swap(b[start], b[i]);
        full_perm(b[n], start + 1);
        swap(b[start], b[i]);
    end
end
```

For the number of addition, since there are $n!$ terms, the number is equal to $n! - 1$.

For the number of multiplication, since there are $n!$ terms in total, each of them is a product of n terms, the number is equal to $n!(n-1)$.

(b)

Algorithm 2: Computing $\det(A)$ by Laplace's method

```
Input: Matrix A [a_ij] of n × n
Output: Determinants of matrix A
Function det(A):
    if n = 1 then
        return a_11;
    end
    row_max_id ← 1;        /*The index of row which has most number of zeros*/
    col_max_id ← 1;        /*The index of column which has most number of zeros*/
    row_max ← 0;           /*The variable used to record the number of zeros in a row*/
    col_max ← 0;           /*The variable used to record the number of zeros in a column*/
    for i ← 1 to n do
        row_sum ← 0;       /*The counter for number of zeros in i^th row*/
        col_sum ← 0;       /*The counter for number of zeros in i^th column*/
        for j ← 1 to n do
            if a_ij = 0 then
                row_sum ← row_sum + 1;
            end
            if a_ji = 0 then
                col_sum ← col_sum + 1;
            end
        end
        if row_sum>row_max then
            row_max ← row_sum;
            row_max_id ← i;
        end
        if col_sum>col_max then
            col_max ← col_sum;
            col_max_id ← i;
        end
    end
    if row_max>col_max then
        sum ← 0;
                           /*Form a new matrix B [b_ij] of (n − 1) × (n − 1)*/
        for i ← 1 to n do
            for row ← 1 to row_max_id-1 do
                for col ← 1 to i-1 do
                    b_{row,col} ← a_{row,col};
                end
                for col ← i+1 to n do
                    b_{row,col−1} ← a_{row,col};
                end
            end
            for row ← row_max_id+1 to n do
                for col ← 1 to i-1 do
                    b_{row−1,col} ← a_{row,col};
                end
                for col ← i+1 to n do
                    b_{row−1,col−1} ← a_{row,col};
                end
            end
            sum ← sum + (−1)^{row_max_id+i} · a_{row_max_id,i} · det(B);
        end
    else
        sum ← 0;
                           /*Form a new matrix B [b_ij] of (n − 1) × (n − 1)*/
        for i ← 1 to n do
            for col ← 1 to col_max_id-1 do
                for row ← 1 to i-1 do
                    b_{row,col} ← a_{row,col};
                end
                for row ← i+1 to n do
                    b_{row−1,col} ← a_{row,col};
                end
            end
            for col ← col_max_id+1 to n do
                for row ← 1 to i-1 do
                    b_{row,col−1} ← a_{row,col};
                end
                for row ← i+1 to n do
                    b_{row−1,col−1} ← a_{row,col};
                end
            end
            sum ← sum + (−1)^{col_max_id+i} · a_{col_max_id,i} · det(B);
        end
    end
    return sum;
end
```

Assume for $n \times n$ matrix, the number of application and multiplication are A_n and

M_n respectively, then

$$A_{n+1} = A_n \cdot (n+1) + n, \quad M_{n+1} = n + 1 + (n+1) \cdot M_n$$

For $\{A_n\}$,

$$\begin{aligned} A_{n+1} = A_n \cdot (n+1) + n &\Rightarrow A_{n+1} + 1 = (n+1)(A_n + 1) \\ &\Rightarrow A_n + 1 = n!(A_1 + 1) \end{aligned}$$

Since $A_1 = 0 = 1! - 1$, $A_n = n! - 1$.

For $\{M_n\}$,

$$\begin{aligned} M_{n+1} = n + 1 + (n+1) \cdot M_n &\Rightarrow \frac{M_{n+1}}{(n+1)!} = \frac{M_n}{n!} + \frac{1}{n!} \\ &\Rightarrow \frac{M_n}{n!} = \sum_{i=1}^{n-1} \frac{1}{i!} + \frac{M_1}{1!} \end{aligned}$$

Since $M_1 = 0$, $M_n = n! \cdot \sum_{i=1}^{n-1} \frac{1}{i!}$.

To sum up, the number of addition is $n! - 1$ and the number of multiplication is $n! \cdot \sum_{i=1}^{n-1} \frac{1}{i!}$.

(c)

Algorithm 3: Computing det(A) by Gauss's method

```
Input: Matrix A [a_ij] of n × n
Output: Determinants of matrix A
Function det(A):
    for k ← 1 to n - 1 do
        /*Try to find a row in which a_lk ≠ 0*/
        l ← k;
        while l <= n do
            if a_lk = 0 then
                l ← l + 1;
            else
                break;
            end
        end
        if l > n then
            return 0;    /*If fail, we have done Gauss elimination for k^th column*/
        else
            /*Exhange row k with row l to make sure a_kk ≠ 0*/
            for i ← 1 to n - 1 do
                swap(a_li, a_ki);
            end
        end
        for i ← k + 1 to n - 1 do
            m ← a_ik/a_kk;
            a_ik ← 0;
            for j ← k + 1 to n do
                a_ij ← a_ij − m · a_kj;
            end
        end
    end
    prod ← 1;
    for i ← 1 to n do
        prod ← prod·a_ii;
    end
    return prod;
end
```

To use Gauss elimination, we need addition with number of

$$\sum_{i=2}^{n}(i-1)\cdot(i-1) = \frac{(n-1)n(2n-1)}{6}$$

and multiplication with number of

$$\sum_{i=2}^{n}(i-1)\cdot i = \frac{n(n+1)(n-1)}{3}$$

To calculate the determinant, no more addition is needed, and $n-1$ times more multiplication is needed. So the number of addition is $\dfrac{(n-1)n(2n-1)}{6}$ and the number of multiplication is $\dfrac{n(n+1)(n-1)}{3}+n-1$.

(d)

Algorithm 4: Computing $\det(A)$ by Dodgson's method

```
Input: Matrix A [a_ij] of n × n
Output: Determinants of matrix A
Function det(A):
    if n = 1 then
        return a_11;
    end
    if n = 2 then
        return a_11 a_22 − a_12 a_21;
    end
    /*make sure at least a_22 ≠ 0*/
    for i ← 1 to n do
        for j ← 1 to n do
            if a_ij ≠ 0 then
                for k ← 1 to n do
                    swap(a_2k, a_ik);
                end
                for k ← 1 to n do
                    swap(a_k2, a_kj);
                end
            end
            break;
        end
    end
    if a_22 = 0 then
        return 0;                                   /*all elements are 0*/
    end
    /*make sure a_k2 ≠ 0 for 2 ⩽ k ⩽ n − 1]*/
    for i ← 3 to n-1 do
        if a_i2 = 0 then
            for k ← 1 to n do
                a_ik ← a_ik + a_2k;
            end
        end
    end
    /*eliminate 0 in the interior of A*/
    for row ← 2 to n - 1 do
        col ← 3;
        while col < n do
            if a_row,col = 0 then
                nscale ← 0;
                for i ← 2 to n-1 do
                    if a_i,col ≠ 0 then
                        if a_i,col−1 / a_i,col < 0 then
                            nscale ← nscale + a_i,col−1 / a_i,col;
                        end
                    end
                end
                nscale ← 1 - nscale;
                for i ← 1 to nn do
                    a_i,col ← nscale·a_i,col−1 + a_i,col;
                end
            end
            col ← col + 1;
        end
    end
    /*Generate the consecutive minors of 2×2 of A, defined as B*/
    for i ← 1 to n-1 do
        for j ← 1 to n-1 do
            b_ij = a_ij · a_i+1,j+1 − a_i,j+1 · a_i+1,j;
        end
    end
    /*Generate the consecutive minors of 2×2 of B, defined as C*/
    for i ← 1 to n-2 do
        for j ← 1 to n-2 do
            c_ij = (b_ij · b_i+1,j+1 − b_i,j+1 · b_i+1,j)/a_i+1,j+1;
        end
    end
    return det (C);
end
```

We ignore the number of addition and multiplication used to eliminate zeros. Then reduce the matrix from $n\times n$ to a single number by using consecutive minors of 2×2, we need addition

$$\sum_{i=1}^{n-1} i^2 \cdot 1 = \frac{(n-1)n(2n-1)}{6}$$

and multiplication

$$\sum_{i=1}^{n-1} i^2 \cdot 2 = \frac{(n-1)n(2n-1)}{3}$$

For $2 \times 2, 3 \times 3, \cdots, (n-2) \times (n-2)$ matrix, we need do division by elements, then it add the number of multiplication for $n \geqslant 4$

$$\sum_{i=2}^{n-2} i^2 = \frac{(n-2)(n-1)(2n-3)}{6} - 1$$

So in total, the number of addition is

$$\frac{(n-1)n(2n-1)}{6}$$

and the number of multiplication is

$$\frac{(n-1)n(2n-1)}{3} + \frac{(n-2)(n-1)(2n-3)}{6} - 1 = \frac{(n-1)(2n^2-3n+2)}{2} - 1 \quad (n \geqslant 4)$$

$$\text{or} \quad \frac{(n-1)n(2n-1)}{3} \quad (n = 1, 2, 3)$$

2.

Dodgson's method trys to eliminate zeros in matrix while Gauss's method creates zeros in lower triangle. In most cases, there are few zeros in a matrix, and therefore no extra efforts is required by using Dodgson's method, i.e. we may ignore the calculation in eliminating zeros. Compared with Gauss's method, we can start our iteration from the very beginning. So people may prefer Dodgson's method.

3.

To use the algorithm, we first need use algorithm GAUSS WITH SCALED PARTIAL PIVOTING, and no more addition or multiplication is needed. So the number of addition is

$$\sum_{k=1}^{n-1} \sum_{i=k+1}^{n} \sum_{j=k+1}^{n} 1 = \sum_{k=1}^{n-1} (n-k-1+1)(n-k-1+1) = \frac{(n-1)n(2n-1)}{6}$$

and the number of multiplication is

$$\sum_{k=1}^{n-1} \Big(\sum_{i=k}^{n} 1 + \sum_{j=k+1}^{n} (1 + \sum_{j=k+1}^{n} 1) \Big) = \sum_{k=1}^{n-1} (n-k+1)^2 = \frac{n(n+1)(2n+1)}{6} - 1$$

4.

(a)

Using block matrix product,

$$\begin{aligned}\mathbf{W} =& \mathbf{AE}+\mathbf{BG}\\ \mathbf{X} =& \mathbf{AF}+\mathbf{BH}\\ \mathbf{Y} =& \mathbf{CE}+\mathbf{DG}\\ \mathbf{Z} =& \mathbf{CF}+\mathbf{DH}\end{aligned}$$

So, 8 matrix multiplications and 4 matrix additions are needed.

(b)

We observed that

1.

$$\begin{aligned}&\mathbf{P}_1+\mathbf{P}_4-\mathbf{P}_5+\mathbf{P}_7\\ =&(\mathbf{A}+\mathbf{D})(\mathbf{E}+\mathbf{H})+\mathbf{D}(\mathbf{G}-\mathbf{E})-(\mathbf{A}+\mathbf{B})\mathbf{H}+(\mathbf{B}-\mathbf{D})(\mathbf{G}+\mathbf{H})\\ =&\mathbf{AE}+\mathbf{AH}+\mathbf{DE}+\mathbf{DH}+\mathbf{DG}-\mathbf{DE}-\mathbf{AH}-\mathbf{BH}+\mathbf{BG}+\mathbf{BH}-\mathbf{DG}-\mathbf{DH}\\ =&\mathbf{AE}+\mathbf{BG}\end{aligned}$$

2.

$$\begin{aligned}&\mathbf{P}_3+\mathbf{P}_5\\ =&\mathbf{A}(\mathbf{F}-\mathbf{H})+(\mathbf{A}+\mathbf{B})\mathbf{H}\\ =&\mathbf{AF}+\mathbf{BH}\end{aligned}$$

3.

$$\begin{aligned}&\mathbf{P}_2+\mathbf{P}_4\\ =&(\mathbf{C}+\mathbf{D})\mathbf{E}+\mathbf{D}(\mathbf{G}-\mathbf{E})\\ =&\mathbf{CE}+\mathbf{DG}\end{aligned}$$

4.

$$\begin{aligned}&\mathbf{P}_1-\mathbf{P}_2+\mathbf{P}_3+\mathbf{P}_6\\ =&(\mathbf{A}+\mathbf{D})(\mathbf{E}+\mathbf{H})-(\mathbf{C}+\mathbf{D})\mathbf{E}+\mathbf{A}(\mathbf{F}-\mathbf{H})+(\mathbf{C}-\mathbf{A})(\mathbf{E}+\mathbf{F})\\ =&\mathbf{AE}+\mathbf{AH}+\mathbf{DE}+\mathbf{DH}-\mathbf{CE}-\mathbf{DE}+\mathbf{AF}-\mathbf{AH}+\mathbf{CE}+\mathbf{CF}-\mathbf{AE}-\mathbf{AF}\\ =&\mathbf{CF}+\mathbf{DH}\end{aligned}$$

So

$$\begin{aligned}\mathbf{W} =& \mathbf{P}_1+\mathbf{P}_4-\mathbf{P}_5+\mathbf{P}_7\\ \mathbf{X} =& \mathbf{P}_3+\mathbf{P}_5\\ \mathbf{Y} =& \mathbf{P}_2+\mathbf{P}_4\\ \mathbf{Z} =& \mathbf{P}_1-\mathbf{P}_2+\mathbf{P}_3+\mathbf{P}_6\end{aligned}$$

(c)

To calculate $\mathbf{P_1} \sim \mathbf{P_7}$, we need 10 additions and 7 multiplications. Using them to calculate $\mathbf{W}, \mathbf{X}, \mathbf{Y}, \mathbf{Z}$, we need 8 additions and 0 multiplications. So in total, we need 18 matrix additions and 7 matrix multiplications by using Strassen's method.

(d)

We see that Strassen's method has saved 1 matrix multiplication compared with calculating directly, which is a quite time-consuming process, especially for large matrix, While, matrix addition is very time-saving compared with it. So Strassen's methods is faster for large matrices

(e)

Assume that by applying Strassen's method recursively, we need $T(n)$ operations (addition/multiplication) to do matrix multiplication for two matrices of $n \times n$, where $n = 2^k$. Then according to algorithm description,

$$\begin{aligned}
T(2n) = & \underbrace{7T(n)}_{\text{matrix multiplication}} + \underbrace{18n^2}_{\text{matrix addition}}, \quad T(1) = 1 \\
\Rightarrow & T(2n) + 6 \cdot (2n)^2 = 7(T(n) + 6n^2), \quad T(1) = 1 \\
\Rightarrow & T(n) = T(2^k) = 7^k(T(1) + 6 \cdot 1^2) - 6n^2 = 7^{\log_2 n + 1} - 6n^2
\end{aligned}$$

and therefore

$$T(n) = 7 \cdot 7^{\log_2 n} - 6n^2 = 7^{\log_2 7 \cdot \log_7 n} - 6n^2 = n^{\log_2 7} - 6n^2 = O(n^{\log_2 7}) \quad \text{as n} \to \infty$$

(f)

```
function [Prod] = Strassen_method(M, N)
%calculate smartix product of M,N by Strassen's method
[row_M,col_M] = size(M);
[row_N,col_N] = size(N);
if (col_M ~= row_N)
    error("false dimension");
end
if (row_M == 1 && col_M == 1 && col_N == 1)
    Prod = M * N;
    return;
end
max_d = max(max(row_M, col_M), col_N);
k = ceil(log(max_d)/log(2));
n = 2^k;
if (row_M ~= n || col_M ~= n) %reform M if necessary
    M(n, n) = 0;
end
if (row_N ~= n || col_N ~= n) %reform N if necessary
    N(n, n) = 0;
end
A = M(1:n/2,1:n/2);
B = M(1:n/2,n/2+1:n);
```

```
C = M(n/2+1:n,1:n/2);
D = M(n/2+1:n,n/2+1:n);
E = N(1:n/2,1:n/2);
F = N(1:n/2,n/2+1:n);
G = N(n/2+1:n,1:n/2);
H = N(n/2+1:n,n/2+1:n);
P1 = Strassen_method(A + D, E + H);
P2 = Strassen_method(C + D, E);
P3 = Strassen_method(A, F - H);
P4 = Strassen_method(D, G - E);
P5 = Strassen_method(A + B, H);
P6 = Strassen_method(C - A, E + F);
P7 = Strassen_method(B - D, G + H);
Prod(1:n/2,1:n/2) = P1 + P4 - P5 +P7;
Prod(1:n/2,n/2+1:n) = P3 + P5;
Prod(n/2+1:n,1:n/2) = P2 + P4;
Prod(n/2+1:n,n/2+1:n) = P1 - P2 + P3 + P6;
end
```

5.

To generate a $n \times n$ matrix $\mathbf{A}$ such that $\mathbf{A}$ is invertible and $\mathbf{A}^{-1}$ as well as $\mathbf{A}$ has only integer elements, we try to apply random Type I and Type III elementary row operations to the unit matrix $\mathbb{I}_{n\times n}$, especially, for type III, we only use integer to do multiplication.

First, it is easy to see that through such kind of method, the generated matrix only contains integers. And elementary row operation is invertible,

$$\mathbf{E}_{i,j} = \mathbf{E}_{i,j}^{-1}, \quad \mathbf{E}_{(\alpha)i,j} = \mathbf{E}_{(-\alpha)i,j}^{-1}$$

we find that the inverse matrices of these transformation matrices also only contain integer. So through this kind of transformation, the generated matrix is invertible, and both of it and its inverse only contain integers.