Algebra makes computation fast, some simple instances

Ranveer[*]

"Beautiful mathematics eventually tends to be useful, and useful mathematics eventually tends to be beautiful"-Anonymous

1 A light warm-up, polynomial factoring

Suppose we ask our computer to compute the following ugly-looking expression where x_is and y_is are some real numbers

$$x_1x_2x_3 + x_1x_2y_3 + x_1y_2x_3 + x_1y_2y_3 + y_1x_2x_3 + y_1x_2y_3 + y_1y_2x_3 + y_1y_2y_3. \tag{1}$$

How many times does our computer bother to compute the basic operations of additions and multiplications? There are 8 product terms and 7 additions. In fact, each product term is the result of 2 multiplications (why?). But hold on, let us recall our school days; the above expression can be written as

$$(x_1 + y_1)(x_2 + y_2)(x_3 + y_3). \tag{2}$$

It says we need only 3 additions and 2 multiplications. But what is a big deal? Here it is. In general, we have

$$(x_1 + y_1)\dots(x_n + y_n) = x_1x_2\dots x_{n-1}x_n + x_1x_2\dots x_{n-1}y_n + x_1x_2\dots y_{n-1}x_n + \cdots + y_1y_2\dots y_{n-1}y_n.$$

To compute LHS we need only n addition and n multiplications (maybe way lesser than n multiplications, how?). What about RHS? There are 2^n product terms to be computed first (why?). How large is 2^n? Take, for example $n = 100$; 2^{100}, is a huge huge huge...may be more than the number of dust particles in this universe, a number practically impossible to compute for any supercomputer in its lifetime. Appreciate the properties of additions and multiplications, the algebra.

2 Finding Fibonacci or Hemachandra numbers

We all know that Fibonacci or Hemachandra numbers are a sequence of numbers, $H_0 = 1, H_1 = 1$, and $H_i = H_{i-1} + H_{i-2}$, for $i \geq 2$ (I am calling them Hemachandra number after listening to an interview and some lectures by the Fields medalist Manjul Bhargava, watch here `https://www.youtube.com/watch?v=siFBqH-LaQQ&t=152s`). The recurrence relation itself gives the following straightforward way to compute n-th

[*]Email: `ranveer@iiti.ac.in`, Copyright:©ranveer, CSE, IIT Indore

Hemachandra number, let it be called Hem(n).

```
int Hem(n)
{
    if(n <= 1){
        return n
    }else{
        return Hem(n-1) + Hem(n-2)
    }
}
```

Listing 1: Calculating n-th Hemchandra number

But how much time will it take to compute Hem(100)? For every $n \geq 2$, Hem(n) is calling its two smaller instances Hem($n-1$), Hem($n-2$). By the time recursion reaches the base cases $n = 0, n = 1$, how many calls to Hem are made? Keep thinking (and find the exact number). Meanwhile, you might have guessed that many times the same computation is done again and again. It is so much again that computing Hem(100) is a practical challenge with this approach as the number of computations grows exponentially with n.
Following is a natural and better approach. Rather than doing the same computation again and again just do it once and save it; whenever you need it, use it, and that's all. Such a strategy is called dynamic programming. Let $H[n]$ be the array that we use to store the values as they are computed. Here is the pseudocode

```
    H[0] = 0
    H[1] = 1
    for( i = 2; i <= n; i = i+1)
    {
        H[i] = H[i-1] + H[i-2]
    }
    return H[n]
```

Listing 2: Calculating n-th Hemchandra number in linear time using dynamic programming

How much time does this process take to compute $H[n]$? Clearly we just need only about n computations $H[2], H[3], \ldots, H[n]$, each just once. It is very fast, in fact, an exponential jump from the previous process. Can we do it even faster? Surprisingly, yes, algebra comes in handy.

2.1 Algebraic way

Using Linear algebra: Let H_n denote the n-th Hemachandra number. Check that we can write

$$\begin{bmatrix} H_{n+1} \\ H_n \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} H_n \\ H_{n-1} \end{bmatrix}.$$

Expanding it we can write

$$\begin{bmatrix} H_{n+1} \\ H_n \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} H_n \\ H_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^2 \begin{bmatrix} H_{n-1} \\ H_{n-2} \end{bmatrix} = \cdots = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n-1} \begin{bmatrix} 1 \\ 0 \end{bmatrix}.$$

So to compute H_n all we need it to compute M^{n-1}, where, $M = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$. How fast we can do it? Do we need to compute matrix multiplication $n-1$ times? No, in fact, we can do it in about $\log_2 n$ multiplications (how?). Again, appreciate the properties of matrix multiplications, the algebra.
If you are a more linear algebraic enthusiast you would have sensed that the above matrix is symmetric and we can get its arbitrary powers just by using its eigenvalues and eigenvectors. The beautiful spectral theorem comes into play, see Theorem 9.3 here https://ranveeriit.github.io/files/BGBB.pdf. For M the eigenvalues are $\lambda_1 = \phi, \lambda_2 = 1-\phi$, where $\phi = \frac{1+\sqrt{5}}{2}$, the golden ratio (ubiquitously present in nature). The eigenvectors are

$$v_1 = \begin{bmatrix} \phi \\ 1 \end{bmatrix}, v_2 = \begin{bmatrix} 1-\phi \\ 1 \end{bmatrix}.$$

As v_1 and v_2 are independent, we can write

$$\begin{aligned} M^{n-1} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n-1} &= \begin{bmatrix} \phi & 1-\phi \\ 1 & 1 \end{bmatrix} \begin{bmatrix} \phi^{n-1} & 0 \\ 0 & (1-\phi)^{n-1} \end{bmatrix} \begin{bmatrix} \phi & 1-\phi \\ 1 & 1 \end{bmatrix}^{-1} \\ &= \begin{bmatrix} \phi & 1-\phi \\ 1 & 1 \end{bmatrix} \begin{bmatrix} \phi^{n-1} & 0 \\ 0 & (1-\phi)^{n-1} \end{bmatrix} \frac{1}{\sqrt{5}} \begin{bmatrix} 1 & \phi-1 \\ -1 & \phi \end{bmatrix}. \end{aligned}$$

Using roots of characteristic equation: For the recurrence relation $H_n = H_{n-2} + H_{n-1}$, the characteristic equation is $x^2 - x - 1 = 0$, which has roots $\phi, 1-\phi$. Solving for H_n using the initial values $H(0) = 0, H(1) = 1$, we get

$$H_n = \frac{\phi^n - (1-\phi)^n}{\sqrt{5}}.$$

Thus to compute H_n all we need to compute is ϕ^n and $(1-\phi)^n$. How fast we can do it?

3 Determinant vs Permanent

$$M = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}.$$

Recall our school days that the determinant of M is

$$a_{11}a_{22}a_{33} - a_{11}a_{23}a_{32} - a_{12}a_{21}a_{33} + a_{12}a_{23}a_{31} + a_{13}a_{21}a_{32} - a_{13}a_{22}a_{31}.$$

The permanent of M is a similar looking quantity where the minus sign is replaced by plus, it gives

$$a_{11}a_{22}a_{33} + a_{11}a_{23}a_{32} + a_{12}a_{21}a_{33} + a_{12}a_{23}a_{31} + a_{13}a_{21}a_{32} + a_{13}a_{22}a_{31}$$

For a square matrix $M = (m_{ij})$ of order n, let $\det M, \operatorname{per} M$ denote the determinant, permanent, respectively. These are given by

$$\det M = \sum_{j=1}^{n} (-1)^{1+j} m_{1j} \det M(1|j),$$

$$\operatorname{per} M = \sum_{j=1}^{n} m_{1j} \operatorname{per} M(1|j),$$

where $M(1|j)$ is the submatrix resulting after deleting the first row and the j-th column of M.
An interesting question is which among determinant and permanent is easier to compute for an $n \times n$ matrix? It looks like permanent should be relatively easier than determinant as we do not need to care about signs. But the reality is extremely counterintuitive. Again take for example $n = 100$, the determinant can be computed very fast whereas permanent will take some lifetime on any supercomputer. Why it is so? The reason is the following neat and beautiful property followed by the determinant but not by the permanent. If $A = BC$ for square matrices A, B, C of the same order, then $\det A = \det B \det C$. Unfortunately, this neat fact does not hold for the permanent. Next, thankfully due to Gaussian elimination an $n \times n$ matrix M can be quickly (in $O(n^{2.367})$) decomposed as

$$M = LUP,$$

where L is a lower triangular matrix and U is an upper triangular matrix, P is a permutation matrix of the same order. The determinant of L and U is the product of their diagonal entries, and the determinant of P is ± 1. Thus we can compute determinant quickly, unfortunately, the similar-looking permanent is extremely extremely difficult to compute, see the introduction part of https://ranveeriit.github.io/files/FCT2023.pdf. In fact, it cannot be done faster (in polynomial time) until P=NP, the famous open conjecture since 1971 https://news.mit.edu/2009/explainer-pnp.