# CS 170 DIS 01

**Released on 2018-09-03**

## 1 Squaring vs multiplying: matrices

The square of a matrix $A$ is its product with itself, $AA$.

(a) Show that five multiplications are sufficient to compute the square of a $2 \times 2$ matrix.

(b) What is wrong with the following algorithm for computing the square of an $n \times n$ matrix?

"Use a divide-and-conquer approach as in Strassen's algorithm, except that instead of getting 7 subproblems of size $n/2$, we now get 5 subproblems of size $n/2$ thanks to part (a). Using the same analysis as in Strassen's algorithm, we can conclude that the algorithm runs in $\Theta(n^{\log_2 5})$ time."

(c) In fact, squaring matrices is no easier than multiplying them. Show that if $n \times n$ matrices can be squared in $\Theta(n^c)$ time, then any $n \times n$ matrices can be multiplied in $\Theta(n^c)$ time.

**Solution:**

a)

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}^2 = \begin{bmatrix} a^2+bc & b(a+d) \\ c(a+d) & bc+d^2 \end{bmatrix}$$

Hence the 5 multiplications $a^2, d^2, bc, b(a+d)$ and $c(a+d)$ suffice to compute the square.

b) We have:

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix}^2 = \begin{bmatrix} A^2+BC & AB+BD \\ CA+DC & CB+D^2 \end{bmatrix} \neq \begin{bmatrix} A^2+BC & B(A+D) \\ C(A+D) & BC+D^2 \end{bmatrix}.$$

We end up getting 5 subproblems that are *not of the same type as the original problem*: We started with a squaring problem for a matrix of size $n \times n$ and three of the 5 subproblems now involve *multiplying* $n/2 \times n/2$ matrices. Hence the recurrence $T(n) = 5T(n/2) + O(n^2)$ does not make sense.

(Also, note that matrices don't commute! That is, in general $BC \neq CB$, so we cannot reuse that computation)

c) Given two $n \times n$ matrices $X$ and $Y$, create the $2n \times 2n$ matrix $A$:

$$A = \begin{bmatrix} 0 & X \\ Y & 0 \end{bmatrix}$$

It now suffices to compute $A^2$, as its upper left block will contain $XY$:

$$A^2 = \begin{bmatrix} XY & 0 \\ 0 & YX \end{bmatrix}$$

Hence, the product $XY$ can be calculated in time $O(S(2n))$. If $S(n) = O(n^c)$, this is also $O(n^c)$.

Note: This is an example of a reduction, and is an important concept that we will see over and over again in this course. We are saying that matrix squaring is no easier than matrix multiplication — because we can trick any program for matrix squaring to actually solve the more general problem of matrix multiplication.

## 2 Find the missing integer

An array $A$ of length $N$ contains all the integers from 0 to $N$ except one (in some random order). In this problem, we cannot access an entire integer in $A$ with a single operation. The elements of $A$ are represented in binary, and the only operation we can use to access them is "fetch the $j$th bit of $A[i]$". Using only this operation to access $A$, give an algorithm that determines the missing integer by looking at only $O(N)$ bits. (Note that there are $O(N \log N)$ bits total in $A$, so we can't even look at all the bits). Assume the numbers are in bit representation with leading 0s.

**Solution:**

(i) **Main idea** Look at least significant bits and compare the 0s and 1s. Discard the numbers whose least significant bit is of the larger set. The bit of the missing number at this position will be the bit of the smaller set. Recursively apply the algorithm and build the missing number at each bit position.

(ii) **Psuedocode**

```
procedure FINDMISSING(A)
    return FINDMISSINGNUM(A, 0)
procedure FINDMISSINGNUM(A, m)
    if length[A] = 0 then
        return m
    B ← values of A with LSB of 0          ▷ LSB stands for least significant bit
    C ← values of A with LSB of 1
    if length[B] ≤ length[C] then
        B ← B with LSB of all numbers removed
        m ← m with 0 prepend to least significant bit
        return FINDMISSINGNUM(B, m)
    else
        C ← C with LSB of all numbers removed
        m ← m with 1 prepend to least significant bit
        return FINDMISSINGNUM(C, m)
```

(iii) **Proof of correctness** Removing a number, $m$, creates an imbalance of 0s and 1s. If $N$ was odd, the number of 0s in least significant bit position should equal the number of 1s. If $N$ was even, then number of 0s should be 1 more than the number of 1s. Thus if all numbers are present, COUNT(0s) $\geq$ COUNT(1s). We have four cases:

- If $LSB$ of $m$ is 0, removing $m$ removes a 0
  - If $N$ is even, COUNT(0s) = COUNT(1s)
  - If $N$ is odd, COUNT(0s) < COUNT(1s)
- If $LSB$ of $m$ is 1, removing $m$ removes a 1
  - If $N$ is even, COUNT(0s) > COUNT(1s)
  - If $N$ is odd, COUNT(0s) > COUNT(1s)

Notice that if COUNT(0s) $\leq$ COUNT(1s) $m$'s least significant bit is 0. We can discard all numbers with LSB of 1 because removing $m$ does not the count of 1s. If COUNT(0s) > COUNT(1s), $m$'s least significant bit is 1. Likewise we can discard all numbers with LSB of 0.
Assume that this condition applies for all bit positions up to $k$. If we look at the $k+1$-th bit position, the condition above holds true. The elements at the $k+1$-th bit position have the same bit at the $k$-th position as $m$ and thus are the only elements we are interested in at the $k+1$ position.

(iv) **Running time analysis** Since we care about the number of bits seen, creating the auxiliary arrays looks at $O(N)$ bits. In the worst case, the problem is reduced in half, so we have the recurrence $T(N) = T(N/2) + O(N)$, which, by master's theorem, gives us $T(N) = O(N)$.
As for runtime, creating the auxiliary arrays and counting the number of 0 bits and 1 bits takes $O(N \log N)$ time because each number has at most $\log N$ bits. In the worst case, the problem is reduced in half, so we have the recurrence $T(N) = T(N/2) + O(N \log N) = O(N \log N)$.

## 3 Complex numbers review

(a) Write each of the following numbers in the form $\rho(\cos\theta + i\sin\theta)$ (for real $\rho$ and $\theta$):

(i) $-\sqrt{3} + i$

(ii) The three third roots of unity

(iii) The sum of your answers to the previous items

(b) Let sqrt($x$) represent one of the complex square roots of $x$, so that $(\text{sqrt}(x))^2 = x$. What are the possible values of sqrt(sqrt($-1$))?

You can use any notation for complex numbers, e.g., rectangular, polar, or complex exponential notation.

**Solution:**

(a) (i) $-\sqrt{3} + i = 2(\cos\frac{5\pi}{6} + i\sin\frac{5\pi}{6})$

(ii) $(\cos 0 + i\sin 0), (\cos\frac{2\pi}{3} + i\sin\frac{2\pi}{3}), (\cos\frac{4\pi}{3} + i\sin\frac{4\pi}{3})$

(iii) 0

(b) $\sqrt{-1} = \pm i$;
$\sqrt{i} = \pm\frac{\sqrt{2}}{2}(1+i)$, $\sqrt{i} = \pm\frac{\sqrt{2}}{2}(1-i)$.

Alternatively, $-1 = \cos\pi + i\sin\pi = \cos 3\pi + i\sin 3\pi$.
So, $\sqrt{\cos\pi + i\sin\pi} = \{(\cos\frac{\pi}{2} + i\sin\frac{\pi}{2}), (\cos\frac{3\pi}{2} + i\sin\frac{3\pi}{2})\}$.
Therefore:
$\sqrt{(\cos\frac{\pi}{2} + i\sin\frac{\pi}{2})} = \sqrt{(\cos\frac{5\pi}{2} + i\sin\frac{5\pi}{2})} = \{(\cos\frac{\pi}{4} + i\sin\frac{\pi}{4}), (\cos\frac{5\pi}{4} + i\sin\frac{5\pi}{4})\}$, and
$\sqrt{(\cos\frac{3\pi}{2} + i\sin\frac{3\pi}{2})} = \sqrt{(\cos\frac{7\pi}{2} + i\sin\frac{7\pi}{2})} = \{(\cos\frac{3\pi}{4} + i\sin\frac{3\pi}{4}), (\cos\frac{7\pi}{4} + i\sin\frac{7\pi}{4})\}$.