Week 7: Divide and Conquer

Agenda:

- Divide and Conquer technique
- Multiplication of large integers
- Exponentiation
- Matrix multiplication

2- Divide and Conquer :

- To solve a problem we can break it into smaller subproblems, solve each one recursively, and then merge the solutions
- Have already seen some examples: Mergesort, Quicksort,
- Here we see two examples that have applications in security of communication (cryptography)

Example 1: Multiplication of large integers :

- Suppose we are dealing with integers that have hundreds of bits (e.g. 256 or 512 bits).
- Such integers are too big to fit into one memory word. Need to deisgn an algorithm for multiplication
- The naive algorithm for *addition* takes $O(n)$ steps if the integers are n bits each.
- For multiplication, the elementary algorithm takes $O(n^2)$ steps.
- Goal: do it faster, i.e. $o(n^2)$.
- Suppose that I and J are the two n bit integers to be multiplied.
- Say $I = w \cdot 2^{n/2} + x$ and $J = y \cdot 2^{n/2} + z$.

$$I = \boxed{\qquad w \qquad\qquad\qquad x \qquad}$$

$$J = \boxed{\qquad y \qquad\qquad\qquad z \qquad}$$

- Now it is easy to see that $I \cdot J = w \cdot y \cdot 2^n + (w \cdot z + x \cdot y)2^{n/2} + xz$.

- To multiply by 2^n only needs to shift-left n bits; each shiftleft takes $O(1)$ time.

- So to multiply by 2^n, and $2^{n/2}$ (for the second term), and add the results: $O(n)$ time.

- We have 4 multiplications of integers of $\frac{n}{2}$ bits each: $w \cdot y$, $w \cdot z$, $x \cdot y$, and $x \cdot z$.

- So, the time required for multiplying I and J is: $T(n) = 4T(\frac{n}{2}) + O(n)$.

- Using master theorem: $T(n) \in \Theta(n^2)$.

- But this is not better than the naive algorithm!! What should we do?

- The bottleneck here is: too many recursive calls; so try to reduce the number of instances of size $\frac{n}{2}$.

- **Observation:** Let $r = (w+x)(y+z) = w \cdot y + (w \cdot z + x \cdot y) + x \cdot y$.

- So r contains all the 4 terms we need to compute $I \cdot J$, but not individually.

- What if we compute $p = w \cdot y$ and $q = x \cdot y$, too? Then we have:

  - $(w \cdot z + x \cdot y) = r - p - q$

  - $w \cdot y = p$

  - $x \cdot y = q$

- So the recursive formula for the time is:

$$T(n) = 3T(\frac{n}{2}) + O(n)$$

- Using Master theorem: $T(n) \in \Theta(n^{\log_2 3})$. Thus:

  Theorem: We can multiply two n bit integers in $O(n^{1.585})$ time.

Example 2: Exponentiation

- Given integers A, g, p, want to compute $g^A \mod p$.

- We saw that this problem has application in cryptograph in CMPUT 272.

- Assume that A is a huge integer with hundreds of bits (e.g. 200 bits).

- The naive algorithm to compute g^A takes g and multiplies it A times.

- If A has a few hundred bits (say 400) this is going to take $\approx 2^{400}$ steps.

- But there is a faster way to compute g^A;
- Observation:
$$g^{24} = (g^{12})^2 = ((g^6)^2)^2 = (((g^3)^2)^2)^2 = (((((g^2 \cdot g)^2)^2)^2$$
- note that taking square of a number needs only one multiplication; this way, to compute g^{24} we need only 5 multiplication instead of 24.

```
Procedure Expon-mod (g, A, p)

if A = 0 then
    return 1
else
    if A is odd then
        a ⟵ Expon-mod (g, A − 1, p)
        return a · g  mod p
    else
        a ⟵ Expon-mod (g, A/2, p)
        return a · a  mod p
```

- Let $T(A)$ be the number of multiplications required to compute $g^A\ \ modp$. For simplicity, assume $A = 2^k$ for some $k \geq 1$.
$$\begin{aligned} T(A) &= T(\frac{A}{2}) + 1 \\ &= T(\frac{A}{4}) + 1 + 1 \\ &\vdots \\ &= T(\frac{A}{2^k}) + k \end{aligned}$$
- Therefore, $T(A) \in O(\log A)$.

Example 3: Matrix multiplication:

- Assume we are given two $n \times n$ matrix X and Y to multiply.
- These are huge matrices, say $n \approx 50,000$.
- The native algorithm will have to multiply one row of X by one column of Z (i.e. $O(n)$ multiplication) to find out only one entry of the result Z
- Total time will be $O(n^3)$.
- Want to use divide and conquer to speed things up; for simplicity assume n is a power of 2.
- Break each of X and Y into 4 submatrices of size $\frac{n}{2} \times \frac{n}{2}$ each:

$$\underbrace{\begin{bmatrix} A & B \\ C & D \end{bmatrix}}_{X} \underbrace{\begin{bmatrix} E & F \\ G & H \end{bmatrix}}_{Y} = \underbrace{\begin{bmatrix} I & J \\ K & L \end{bmatrix}}_{Z}$$

- Therefore:

$$\left.\begin{array}{l} I = AE + BG \\ J = AF + BH \\ K = CE + DG \\ L = CF + DH \end{array}\right\} \longrightarrow$$

need 8 multiplications of subproblems of size $\frac{n}{2}$ each

- We also need to spend $O(n^2)$ time to add up these results.

Matrix multiplication (cont'd):

- If $T(n)$ is the time to multiply two matrices of size $n \times n$ each, then:
$$T(n) = 8T(\frac{n}{2}) + O(n^2)$$

- Using master theorem: $T(n) \in \Theta(n^{\log_2 8}) = \Theta(n^3)$.

- So this is as bad as the naive algorithm. No improvement yet.

- We use an idea similar to the one for multiplication of large integers: reduce the number of subproblems using a clever trick.

- compute the following 7 multiplications (each consisting of two subproblems of size $\frac{n}{2}$ each):
$$\begin{aligned} S_1 &= A(F - H) \\ S_2 &= (A + B)H \\ S_3 &= (C + D)E \\ S_4 &= D(G - E) \\ S_5 &= (A + D)(E + H) \\ S_6 &= (B - D)(G + H) \\ S_7 &= (A - C)(E + F) \end{aligned}$$

- Then:
$$\begin{aligned} I &= S_5 + S_6 + S_4 - S_2 \\ &= (A + D)(E + H) + (B - D)(G + H) + D(G - E) - (A + B)H \\ &= AE + DE + AH + DH + BG - DG + BH - DH + \\ &\qquad DG - DE - AH - BH \\ &= AE + BG \end{aligned}$$

Matrix multiplication (cont'd):

- Similarly, it can be verified easily that:

$$\begin{aligned} J &= S_1 + S_2 \\ K &= S_3 + S_4 \\ L &= S_1 - S_7 - S_3 + S_5 \end{aligned}$$

- So to compute I, J, K, and L, we only need to compute $S_1, \ldots, S_7$; this requires solving seven subproblems of size $\frac{n}{2}$, plus a constant (at most 16) number of addition each taking $O(n^2)$ time.

$$T(n) = 7T(\frac{n}{2}) + O(n^2)$$

- Using master theorem and since $\log_2 7 \approx 2.808$:

$$T(n) \in O(n^{2.808})$$

- For $n = 50,000$: $n^3 \approx 10^{17}$ and $n^{2.808} \approx 10^{13}$; $\longrightarrow$ this algorithm is about 10,000 times faster than the naive algorithm.