CSC373 Algorithms

Jonah Chen

1 Divide and Conquer

- Divide and Conquer algorithm:
  1. Divide problem of size n into a smaller subproblems of size n/b each
  2. Recursively solve each subproblem
  3. Combine the subproblem solutions into the solution of the original problem
- Runtime: $n > 1 : T(n) = aT(n/b) + cn^d; n = 1 : T(1) = c$
- Master Theorem: T(n) depends on relation between a and b^d.

$$\begin{cases} a < b^d : T(n) = \Theta(n^d) \\ a = b^d : T(n) = \Theta(n^d \log n) \\ a > b^d : T(n) = \Theta(n^{\log_b a}) \end{cases} \tag{1}$$

  – Note that the running time does not depend on the constant c
  – In many algorithms $d = 1$ (combining take linear time)
- Examples:
  – Merge sort — sorting array of size n ($a = 2,\ b = 2,\ d = 1 \to a = b^d$) so $T(n) = \Theta(n \log n)$
  – Binary search — searching sorted array of size n ($a = 1,\ b = 2,\ d = 0 \to a = b^d$) so $T(n) = \Theta(\log n)$

1.1 Karatsuba Multiplication

- **Add** two binary n-bit numbers naively takes $\Theta(n)$ time
- **Multiply** two binary n-bit numbers naively takes $\Theta(n^2)$ time
- Divide and Conquer approaches
  1. Multiply x and y. We can divide them into two parts

$$x = x_1 \cdot 2^{n/2} + x_0 \tag{2}$$
$$y = y_1 \cdot 2^{n/2} + y_0 \tag{3}$$
$$x \cdot y = x_1 \cdot y_1 \cdot 2^n + (x_1 \cdot y_0 + x_0 \cdot y_1) \cdot 2^{n/2} + x_0 \cdot y_0 \tag{4}$$

     – $T(n) = 4T(n/2) + cn; T(1) = c$
     – $a = 4, b = 2, d = 1$ Master Theorem case 3, $T(n) = \Theta(n^{\log_2 4}) = \Theta(n^2)$.
     – This is the same complexity of the naive approach, making this approach useless.
  2. Reconsider (4), we may rewrite $(x_1 \cdot y_0 + x_0 \cdot y_1)$ as $(x_1 + x_0) \cdot (y_1 + y_0) - x_1 \cdot y_1 - x_0 \cdot y_0$

$$x \cdot y = x_1 \cdot y_1 \cdot 2^n + ((x_1 + x_0) \cdot (y_1 + y_0) - x_1 \cdot y_1 - x_0 \cdot y_0) \cdot 2^{n/2} + x_0 \cdot y_0 \tag{5}$$

- $T(n) = 3T(n/2) + cn; T(1) = c$
- $a = 3, b = 2, d = 1$, Master Theorem case 3, $T(n) = \Theta(n^{\log_2 3}) = \Theta(n^{\log_2 3}) \approx \Theta(n^{1.585})$
  - Minor issue: a carry may increase $x_1 + x_0$ and $y_1 + y_0$ to $\frac{n}{2} + 1$. We can easily prove this by isolating the carry bit and reevaluating the complexity.

- To deal with integers which doesn't have a power of 2 number of bits, we can pad the numbers with 0s to make them have a power of 2 number of bits. This may at most increase the complexity by 3x.
- 1971: $\Theta(n \cdot \log n \cdot \log\log n)$
- 2019: Harvey and van der Hoeven $\Theta(n \log n)$. We do not know if this is optimal.

1.2 Strassen's MatMul Algorithm

- Let A and B be two $n \times n$ matrices (for simplicity n is a power of 2), we want to compute $C = AB$.
- The naive approach takes $\Theta(n^3)$ time.
  1. Divide A and B into 4 submatrices of size $\frac{n}{2} \times \frac{n}{2}$ each

$$A = \begin{bmatrix} A_1 & A_2 \\ A_3 & A_4 \end{bmatrix}. \tag{6}$$

     Then, C can be calculated with

$$C_1 = A_1 B_1 + A_2 B_3 \tag{7}$$
$$C_2 = A_1 B_2 + A_2 B_4 \tag{8}$$
$$C_3 = A_3 B_1 + A_4 B_3 \tag{9}$$
$$C_4 = A_3 B_2 + A_4 B_4 \tag{10}$$

     - $T(n) = 8T(n/2) + cn^2; T(1) = c$
     - $a = 8, b = 2, d = 2$, case 3 $T(n) = \Theta(n^{\log_2 8}) = \Theta(n^3)$
  2. **Idea:** Compute C_1, C_2, C_3, C_4 with only 7 multiplications, not 8.

$$M_1 = (A_2 - A_4)(B_3 + B_4) \tag{11}$$
$$M_2 = (A_1 + A_4)(B_1 + B_4) \tag{12}$$
$$M_3 = (A_1 - A_3)(B_1 + B_2) \tag{13}$$
$$M_4 = (A_1 + A_2)B_4 \tag{14}$$
$$M_5 = A_1(B_2 - B_4) \tag{15}$$
$$M_6 = A_4(B_3 - B_1) \tag{16}$$
$$M_7 = (A_3 + A_4)B_1 \tag{17}$$

     With these we can compute C_1, C_2, C_3, C_4 with only additions of the M matrices.

$$C_1 = M_1 + M_2 - M_4 + M_6 \tag{18}$$
$$C_2 = M_4 + M_5 \tag{19}$$
$$C_3 = M_6 + M_7 \tag{20}$$
$$C_4 = M_2 - M_3 + M_5 + M_7 \tag{21}$$

     - $T(n) = 7T(n/2) + cn^2; T(1) = c$
     - $a = 7, b = 2, d = 2$, case 3 $T(n) = \Theta(n^{\log_2 7}) = \Theta(n^{\log_2 7}) \approx \Theta(n^{2.807})$

- If n is not a power of 2, we zero-pad the matrices to have n as a power of two. This may increase the complexity by at most a factor of 7.