Master Theorem

Reading: Goodrich/Tamassia §11.1.1

It is common for a divide-and-conquer algorithm’s running time to have a recurrence relation of the following form:

\[ T(n) = aT(n/b) + f(n), \]

for some \( a \geq 1, \) \( b > 1, \) and \( f(n) \) is asymptotically positive.

1. If there is a small constant \( \varepsilon > 0 \) such that \( f(n) = \mathcal{O}(n^{\log_b a - \varepsilon}) \), then \( T(n) = \Theta(n^{\log_b a}) \)

2. If there is a constant \( k \geq 0 \) such that \( f(n) = \Theta(n^{\log_b a \log k n}) \), then \( T(n) = \Theta(n^{\log_b a \log k + 1} n) \)

3. If there is a small constant \( \varepsilon > 0 \) such that \( f(n) = \Omega(n^{\log_b a + \varepsilon}) \), then \( T(n) = \Theta(f(n)) \).

Using the Master Theorem

Use the Master Theorem to solve the following:

1. \( T(n) = 4T(n/2) + n \)
2. \( T(n) = 2T(n/2) + n \log n \)
3. \( T(n) = T(n/3) + n \)
4. \( T(n) = 9T(n/3) + n^{2.5} \)

Using the Master Method

After we cover the Master Method, consider doing these as extra practice.

5. \( T(n) = 2T(n/2) + 1 \)
6. \( T(n) = 2T(n/2) + n \)
7. \( T(n) = 2T(n/2) + n^2 \)
8. \( T(n) = 2T(n/4) + 1 \)
9. \( T(n) = 2T(n/4) + \sqrt{n} \)
10. \( T(n) = 2T(n/4) + n \)
11. \( T(n) = 9T(n/3) + n \)
12. \( T(n) = T(2n/3) + 1 \)
13. \( T(n) = 3T(n/4) + n \log n \)
14. \( T(n) = 2T(n/4) + n^2 \)
15. \( T(n) = 2T(n/4) + n^4 \)
16. \( T(n) = T(7n/10) + n \)
17. \( T(n) = 6T(n/4) + n^2 \)
18. \( T(n) = 7T(n/3) + n^2 \)
19. \( T(n) = 7T(n/2) + n^2 \)

\(^1\)Technically, it must also be the case that \( af(n/b) \leq \delta f(n) \) for some constant \( \delta < 1 \) and for all sufficiently large \( n \). I will not give you any recurrence relations in CompSci 161 that fail to meet this condition.

© CompSci 260P Fall 2022– Michael Shindler– University of California, Irvine
This handout may not be reposted without the express written permission of the professor teaching this course.
Integer Multiplication

Given two \( n \)-bit integers \( X \) and \( Y \), compute \( X \times Y \). The algorithm you learned for this in grade school takes time \( O(n^2) \).

For our divide-and-conquer algorithm, we are going to divide \( X \) and \( Y \) each into their “higher order” and “lower order” bits first; \( X_H \) is the \( n/2 \) higher-order bits, and \( X_L \) is the lower-order bits.

**Example** If \( X = 156 = 10011100 \) and \( Y = 225 = 11100001 \), then:

\[
\begin{array}{c|c|c|c}
X_H & X_L & Y_H & Y_L \\
1001 & 1100 & 1110 & 0001 \\
\end{array}
\]

Note that \( X = X_H \times 2^{n/2} + X_L \) and \( Y = Y_H \times 2^{n/2} + Y_L \)

**Initial Algorithm** Using algebra, we can see that

\[
X \times Y = (X_H \times 2^{n/2} + X_L) \times (Y_H \times 2^{n/2} + Y_L) \\
= X_H \cdot Y_H \times 2^n + (X_H Y_L + X_L Y_H) \times 2^{n/2} + X_L Y_L
\]

**Finish the Algorithm:**

Algorithm Mult\((X, Y)\)

Create \( X_H, X_L, Y_H, Y_L \)

\( A = \text{Mult}(X_H, Y_H) \)

**Question 1.** That’s four recursive calls, each of size \( n/2 \), plus some addition, which takes an additional \( O(n) \) time. Why isn’t this a good algorithm for computing \( X \times Y \)? Can we do better?
Strassen’s Algorithm (Time Permitting)

Reading: G/T §11.3. In algebra, you saw an algorithm to multiply two $n \times n$ matrices,

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

- $I = AE + BG$
- $J = AF + BH$
- $K = CE + DG$
- $L = CF + DH$

If the matrices are $2 \times 2$, then $A \ldots L$ are elements directly and we can compute each as per the right-hand side. If they are not, we can form the product matrix anyway. Each entry in the product matrix is the result of a dot product of the appropriate row from one matrix and column from the other. Because each dot product takes $\Theta(n^3)$ to compute, the end result is a $\Theta(n^3)$ time brute force algorithm.

Alternately, we can use a divide and conquer algorithm by treating each $n/2 \times n/2$ quadrant as a matrix and performing matrix multiplication instead of scalar multiplication.

**Question 2.** What is the running time of the second approach?

**Question 3.** Adding two matrices takes $\Theta(n^2)$ time. There’s also another reason we should not expect to find an algorithm that isn’t $\Omega(n^2)$ to solve this problem. What is it?

Strassen’s Algorithm computes the resulting matrix in a different way than the straight-forward approach described above.

First, compute $S_1 \ldots S_7$:

- $S_1 = A(F - H)$
- $S_2 = (A + B)H$
- $S_3 = (C + D)E$
- $S_4 = D(G - E)$
- $S_5 = (A + D)(E + H)$
- $S_6 = (B - D)(G + H)$
- $S_7 = (A - C)(E + F)$

Second, compute $I, J, K, L$:

- $I = S_5 + S_6 + S_4 - S_2 = (A + D)(E + H) + (B - D)(G + H) + D(G - E) - (A + B)H = AE + BG$
- $J = S_1 + S_2 = A(F - H) + (A + B)H = AF - AH + AH + BH = AF + BH$
- $K = S_3 + S_4$
- $L = S_1 - S_7 - S_3 + S_5$
Closest Pair of Points

Reading: Goodrich/Tamassia §22.4. Suppose we have \( n \) points, each of which has an x-coordinate \( x_i \) and a y-coordinate \( y_i \). Our goal is to find the pair of points \( p_i \) and \( p_j \) that are closest together. The distance between two points is \( d(p_i, p_j) \).

Here is a Brute-Force approach to this problem:

**Closest-Pair**

**Input:** \( n \) points in 2D-space

**Output:** The closest pair of points.

```plaintext
min = \infty
for i = 2 \to n do
    for j = 1 \to i - 1 do
        if \((x_j - x_i)^2 + (y_j - y_i)^2\) < min then
            min = \((x_j - x_i)^2 + (y_j - y_i)^2\)
            closestPair = \(((x_i, y_i), (x_j, y_j))\)
        return closestPair
```

What is the running time of this algorithm?

To improve on the running time of the brute-force algorithm, we can try to set up our usual start for divide and conquer. For convenience, let’s assume the points are sorted by y-coordinate before we first call this algorithm. We can do this in \( O(n \log n) \) time first; if the eventual running time is \( \Omega(n \log n) \), this won’t matter, and if we achieve \( o(n \log n) \) for the rest of the algorithm, this will dominate the running time.

**Closest-Pair**

**Input:** \( n \) points in 2D-space

**Output:** The closest pair of points.

- If \( P \) is sufficiently small, use brute force. // \( O(1) \)
- \( x_m \leftarrow \) median x-value from \( P \)
- \( L \leftarrow \) any points from \( P \) with x-coordinate \( \leq x_m \)
- \( R \leftarrow \) any points from \( P \) with x-coordinate \( x_m \)
- Let \( l_1 \) and \( l_2 \) be the closest pair of points in \( L \), found recursively.
- Let \( r_1 \) and \( r_2 \) be the closest pair of points in \( R \), found recursively.
- return whichever pair is closer together // Incorrect but good starting point.

The above algorithm is clearly incorrect; why?

How do we fix it?

How do we fix it while having a better running time than the brute force algorithm?