The Divide-and-Conquer technique. Karatsuba's and Strassen's algorithms.

Overview

We have seen several problems so far whose solutions look similar. This week we introduce a general algorithmic technique called *divide-and-conquer*. It is a powerful tecnique which yields elegant solutions to many problems. We'll explore this technique by seeing it in action on a couple of problems.

1 Divide-and-conquer

Let's assume, generically, that we want to solve a problem P. A *divide-and-conquer* (DC) solution for P looks as follows:

```
divideAndConquer(P)
1  // Input: problem P
2  // Output: solution of P
3  if size of P is small
4        solve it (e.g. brute force) and return solution
5  else
6        Divide: divide P into smaller problems P_1, P_2
7        sol1 = divideAndConquer(P_1)
8        sol2= divideAndConquer(P_1)
9        Conquer/Combine: combine solutions to P_1, P_2 into solution for P.
```

2 Example 1: Mergesort

- We've already seen a divide-and-conquer algorithm: Mergesort
- Let's review it and see how it fits the general divide-and-conquer framework

```
Mergesort(A)
1  // Input: array A
2  // Output: sorted array
3  if A.size() == 1
4       return A (one element is already sorted)
5  else
6       Divide: divide A into two halves A_1 and A_2
7       sorted_A1 = Mergesort(A_1)
8       sorted_A2 = Mergesort(A_2)
9       Combine: merge sorted_A1 and sorted_A2 into one sorted array and return it
```

- Analysis: $T(n) = 2T(n/2) + \Theta(n)$, which solves to $T(n) = \Theta(n \lg n)$

3 Example 2: Multiplying large integers

- The problem: We want to write an algorithm to multiply arbitrarily large integers.
- Example: $X = 13519384653184763746$ and $Y = 32875641827561875665$
- The primitive type `int` in a programming language has a fixed precision — usually 4 bytes (32 bits). For signed values, one of the bits is used for the sign. For unsigned values, all bits are used for the value. In this case, if all the 32 bits are used for the value, the largest value representable on 32 bits is 11111111...11 which is $1 + 2 + 2^2 + 2^3 + ... + 2^{31} = 2^{32} - 1$ which is approx. 4 billion. So the largest integer value you can store as the primitive type `int` is (roughly) $4 \cdot 10^9$.
- There are special applications (e.g. cryptography) that need to handle very large integers with thousands and tens of thousands of digits. These applications use special libraries for large integers, which represent the integers as arrays of digits rather than `int`s. These libraries implement functions for the usual arithmetic and logic operations on integers, such as addition, subtraction, division, multiplication, and more. Today we are looking at *multiplying* two large integers.
- So, we are given two arrays X and Y each representing the digits of a very large integer. For simplicity we will assume that the size of the two arrays is the same, and we'll call it n as usual.
- We want to develop an algorithm to compute the product of X and Y, $Z = X \cdot Y$
- The product of two n-digit numbers can have up to $2n$ digits (e.g. $9 \cdot 9 = 81$), so Z will be an array of size $2n$
- How do you compute Z? Let's see an example.
- Example: $X = 357$, $Y = 125$. We could do what we all learnt in school, multiply each digit in X by each digit in Y, and then add the results. Let's assume that the addition/multiplication

of two one-digit integers takes $\Theta(1)$ time. When X and Y have n digits each, the procedure we learnt in school consists of performing $\Theta(n^2)$ additions and $\Theta(n^2)$ multiplication, which overall is $\Theta(n^2)$ time. Right?

- With our "algorithms hat" on, we ask the usual question: Can we do better than quadratic?

3.1 Towards a divide-and-conquer approach

- Let's try a divide-and-conquer approach. Our problem is multiplying two n-digit integers. A half-problem woud be multiplying integers represented on $n/2$-digits. We would need to frame the multiplication of two n-digit integers in terms of multiplying two $n/2$-digit integers. Let's see.

- Example: $X = 1427$ and $Y = 3659$, with $n = 4$ digits. Let's split X and Y into two halves: $X = 1400 + 27 = 14 \cdot 10^2 + 27$, and $Y = 3600 + 59 = 36 \cdot 10^2 + 59$. Right?

- So we can say that $X \cdot Y = (14 \cdot 10^2 + 27) \cdot (36 \cdot 10^2 + 59) = 14 \cdot 36 \cdot 10^4 + (27 \cdot 36 + 59 \cdot 14) \cdot 10^2 + 59 \cdot 27$. We expressed the product of two 4-digit numbers in terms of four products of 2-digit numbers, and 3 additions of 4-digit numbers. We're getting there!

- So far we have used base-10, because we are humans. But computers use base-2. Everything is the same as in base 10, except with 2 instead of 10. For e.g. $1001_2 = (10 \cdot 2^2 + 01)_2 = (2 \cdot 2^2 + 1)_{10} = (2^3 + 1)_{10} = 9_{10}$. If base-2 is awkward, you can mentally replace 2 with 10 everywhere.

- Let's generalize: Let's denote by A the first half of X and by B the second half of X, that is: $X = [AB]$, where $A = [X_0, X_1, ...X_{n/2}]$ and $B = [X_{n/2+1}, ..., X_{n-1}]$. We get that

$$X = A \cdot 2^{n/2} + B$$

- Similarly, Let's denote by C the first half of Y and by D the second half of Y, that is: $Y = [CD]$, where $C = [Y_0, Y_1, ...Y_{n/2}]$ and $D = [Y_{n/2+1}, ..., Y_{n-1}]$. We get that

$$Y = C \cdot 2^{n/2} + D$$

- Now we can write

$$X \cdot Y = (A \cdot 2^{n/2} + B) \cdot (C \cdot 2^{n/2} + D)$$

- Opening the parenthesis we get:

$$X \cdot Y = A \cdot C \cdot 2^n + (A \cdot D + B \cdot C) \cdot 2^{n/2} + B \cdot D$$

- What does this mean? To compute $X \cdot Y$, the product of two n-digit numbers, we need to compute:

  1. We need to compute 4 products of two $n/2$-digit numbers: $A \cdot C$, $A \cdot D$, $B \cdot C$ and $A \cdot D$.
  2. We need to compute three sums of $\Theta(n)$-digit numbers (there are three "+" signs in the expression above). Xdding two n-digit numbers can be done in $\Theta(n)$-time, by using the obvious algorithm (add two digits one at a tiime, going from right to left).

3. Multiplying by a power of 2 (in base-2) means shifting to the left that many bits, and adding trailing 0s. That can be done in linear time in terms of the number of bits in our number and the exponent. So multiplying $A \cdot C$ by 2^n runs in the total number of bits in $A \cdot C$ plus n, which is $\Theta(n)$.
4. Similarly, multiplying $(A \cdot D + B \cdot D)$ by $2^{n/2}$ runs in $\Theta(n)$ time.

- So overall we expressed $X \cdot Y$ as four products of $n/2$-digit numbers; once these products are known, it takes $\Theta(n)$ work to figure out $X \cdot Y$.
- Let $T(n)$ be the running time for computing the product of two n-digit numbers. Then we can write that $T(n) = 4T(n/2) + \Theta(n)$
- The recurrence solves to $\Theta(n^2)$ time
- Really?
- Really. We get the same quadratic time as with the "straightforward" algorithm! Worse actually, because of recursion overhead.

3.2 Karatsuba's idea

- Still, the idea can be used to get a better algorithm.
- How? If we could express $X \cdot Y$ in terms of only **three** products of $n/2$-digit numbers, We would get the recurrence $T(n) = 3T(n/2) + \Theta(n)$, which solves to $\Theta(n^{\lg 3}) = n^{1.584} << n^2$
- But... How ??!
- Remember that
$$X \cdot Y = A \cdot C \cdot 2^n + (A \cdot D + B \cdot C) \cdot 2^{n/2} + B \cdot D$$
We need $A \cdot C$, $A \cdot D + B \cdot C$ and $B \cdot D$
- Karatsuba observed that we can compute $A \cdot D + B \cdot C$ in terms of the other two products but with some additional additions/subtractions:
$A \cdot D + B \cdot C = (A + B) \cdot (C + D) - A \cdot C - B \cdot D$
- Therefore we would need only three products:
$A \cdot B, C \cdot D$ and $(A + B) \cdot (C + D)$
- The algorithm:

IntegerMultiply (X, Y):

- Divide X and Y into halves: A, B, C, D
- Compute three $n/2$-digit products recursively, namely let $Z_1 = A \cdot C, Z_2 = B \cdot D$ and $Z_3 = (A + B) \cdot (C + D)$
- Combine results by doing a bunch of additions and subtractions and shifts, namely
  $Z_3 = Z_3 - Z_1 - Z_2$
  $Z = Z_1 \cdot 2^n + Z_3 \cdot 2^{n/2} + Z_2$
- Return Z as the result

- We get the recurrence $T(n) = 3T(n/2) + \Theta(n)$, which solves to $\Theta(n^{\lg 3}) = n^{1.584}$
- Self-study exercise: Consider two numbers on 4 digits each, and compute their product using Karatsuba's algorithm; use base-10 for simplicity.

Comments: Large integer multiplication

A couple of algorithms have been developped which improve on Karatsuba; there is an algorithm by Schonhage and Strassen (1971) that runs in $O(n \lg n \lg \lg n)$; in 2007 a new algorithm was published which has a theoretically better upper bound of $O(n \lg n \cdot 2^{O(\log^* n)})$; several improvements to this algorithm were published in the last 10 years, culminating with an $O(n \lg n)$ algorithm proposed in 2019 by Harvey and Van Der Hoeven; because Strassen conjectured that $\Omega(n \lg n)$ is a lower bound, this last algorithm is believed to be optimal. These algorithms are faster than Karatsuba for very very very very very large values of n. For small values of n Karatsuba is faster.

4 Example 3: Matrix Multiplication

Let X and Y be two $n \times n$ matrices:

$$X = \begin{Bmatrix} x_{11} & x_{12} & \cdots & x_{1n} \\ x_{21} & x_{22} & \cdots & x_{1n} \\ x_{31} & x_{32} & \cdots & x_{1n} \\ \cdots & \cdots & \cdots & \cdots \\ x_{n1} & x_{n2} & \cdots & x_{nn} \end{Bmatrix}$$

$$Y = \begin{Bmatrix} y_{11} & y_{12} & \cdots & y_{1n} \\ y_{21} & y_{22} & \cdots & y_{1n} \\ y_{31} & y_{32} & \cdots & y_{1n} \\ \cdots & \cdots & \cdots & \cdots \\ y_{n1} & y_{n2} & \cdots & y_{nn} \end{Bmatrix}$$

We want to compute the product $Z = X \cdot Y$, which is defined as $z_{ij} = \sum_{k=1}^{n} X_{ik} \cdot Y_{kj}$

- **The straightfoward algorithm:** For every i=1 to n, for every j=1 to n, compute z_{ij} as the sum $\sum_{k=1}^{n} X_{ik} \cdot Y_{kj}$
- Analysis: There are n^2 elements, and each one needs a loop to calculate $\Rightarrow n^2 \cdot n = \Theta(n^3)$
- Can we do better? That is, is it possible to multiply two matrices faster than $\Theta(n^3)$?
- This was an open problem for a long time... until Strassen came up with an algorithm in 1969. His idea was to use divide-and-conquer.

4.1 Towards matrix multiplication via divide-and-conquer

- Let's imagine that n is a power of two. We can view each matrix as consisting of four $n/2$-by-$n/2$ matrices.

$$X = \begin{Bmatrix} A & B \\ C & D \end{Bmatrix}, Y = \begin{Bmatrix} E & F \\ G & H \end{Bmatrix}$$

- Their product $X \cdot Y$ can be written as:

$$\begin{Bmatrix} A & B \\ C & D \end{Bmatrix} \cdot \begin{Bmatrix} E & F \\ G & H \end{Bmatrix} = \begin{Bmatrix} (A \cdot E + B \cdot G) & (A \cdot F + B \cdot H) \\ (C \cdot E + D \cdot G) & (C \cdot F + D \cdot H) \end{Bmatrix}$$

- This leads to a divide-and-conquer solution:

> MatrixMultiply (X, Y):
> - Divide X and Y into eight sub-matrices A, B, C, D, E, F, G, H.
> - Compute eight $n/2$-by-$n/2$ matrix multiplications recursively, namely $A \cdot E, B \cdot G, A \cdot F, B \cdot H, C \cdot E, D \cdot G, C \cdot F, D \cdot H$
> - Combine results (by doing 4 matrix additions) and copy the results into a matrix Z
> - Return matrix Z as the result

ANALYSIS:

- Adding two n-by-n matrices runs in $\Theta(n^2)$ time.
- The running time is given by $T(n) = 8T(n/2) + \Theta(n^2)$, which solves to $T(n) = \Theta(n^3)$
- Cool idea, but not so cool result.....since we already discussed that the straightforward algorithm runs in $O(n^3)$
- Can we do better?

4.2 Strassen's matrix multiplication algorithm

- Strassen's algorithm is based on the following observation:

  The recurrence

$$T(n) = 8T(n/2) + \Theta(n^2) \Rightarrow T(n) = \Theta(n^3)$$

while the recurrence

$$T(n) = 7T(n/2) + \Theta(n^2) \Rightarrow T(n) = \Theta(n^{\lg 7})$$

- Strassen found a very clever way to express $X \cdot Y$ in terms of only **seven** products of $n/2$-by-$n/2$ matrices
- With same notation as before, we define the following seven $n/2$-by-$n/2$ matrices:

$$\begin{aligned}
S_1 &= (B - D) \cdot (G + H) \\
S_2 &= (A + D) \cdot (E + H) \\
S_3 &= (A - C) \cdot (E + F) \\
S_4 &= (A + B) \cdot H \\
S_5 &= A \cdot (F - H) \\
S_6 &= D \cdot (G - E) \\
S_7 &= (C + D) \cdot E
\end{aligned}$$

- Strassen observed that we can write the product Z as:

$$Z = \begin{Bmatrix} A & B \\ C & D \end{Bmatrix} \cdot \begin{Bmatrix} E & F \\ G & H \end{Bmatrix} = \begin{Bmatrix} (S_1 + S_2 - S_4 + S_6) & (S_4 + S_5) \\ (S_6 + S_7) & (S_2 + S_3 + S_5 - S_7) \end{Bmatrix}$$

- For e.g. let's test that $S_6 + S_7$ is really $C \cdot E + D \cdot G$

$$\begin{aligned}
S_6 + S_7 &= D \cdot (G - E) + (C + D) \cdot E \\
&= D \cdot G - D \cdot E + C \cdot E + D \cdot E \\
&= D \cdot G + C \cdot E
\end{aligned}$$

- This leads to a divide-and-conquer algorithm:

> StrassenMM(X, Y):
> - Divide X and Y into 8 sub-matrices A, B, C, D, E, F, G, H.
> - Compute $S_1, S_2, S_3, ..., S_7$. This step involves 10 matrix additions and 7 multiplications (which are computed recursively).
> - Compute $S_1 + S_2 - S_4 + S_6, S_4 + S_5, S_6 + S_7$ and $S_2 + S_3 + S_5 - S_7$ and copy them in Z. This step involves 8 additions/subtractions of $n/2$-by-$n/2$ matrices.

ANALYSIS:

- All additions/subtractions/copying can be done in $\Theta(n^2)$ time
- Overall there are (only) 7 recursive calls

- The running time is given by $T(n) = 7T(n/2) + \Theta(n^2)$, which solves to $O(n^{\lg 7})$.
- Lets solve the recurrence using the iteration method

$$\begin{aligned}
T(n) &= 7T(n/2) + n^2 \\
&= n^2 + 7(7T(\frac{n}{2^2}) + (\frac{n}{2})^2) \\
&= n^2 + (\frac{7}{2^2})n^2 + 7^2T(\frac{n}{2^2}) \\
&= n^2 + (\frac{7}{2^2})n^2 + 7^2(7T(\frac{n}{2^3}) + (\frac{n}{2^2})^2) \\
&= n^2 + (\frac{7}{2^2})n^2 + (\frac{7}{2^2})^2 \cdot n^2 + 7^3T(\frac{n}{2^3}) \\
&= n^2 + (\frac{7}{2^2})n^2 + (\frac{7}{2^2})^2n^2 + (\frac{7}{2^2})^3n^2 .... + (\frac{7}{2^2})^{\lg n-1}n^2 + 7^{\lg n} \\
&= \sum_{i=0}^{\lg n-1} (\frac{7}{2^2})^i n^2 + 7^{\lg n} \\
&= n^2 \cdot \Theta((\frac{7}{2^2})^{\lg n-1}) + 7^{\lg n} \\
&= n^2 \cdot \Theta(\frac{7^{\lg n}}{(2^2)^{\lg n}}) + 7^{\lg n} \\
&= n^2 \cdot \Theta(\frac{7^{\lg n}}{n^2}) + 7^{\lg n} \\
&= \Theta(7^{\lg n}) \\
&= n^{\lg 7}
\end{aligned}$$

  So the solution is $T(n) = \Theta(n^{\lg 7}) = \Theta(n^{2.81...})$

Matric multiplication: Some comments

- **In practice:** Strassen's algorithm "hides" a much bigger constant in $\Theta()$ than the straightfoward cubic algorithm, and is efficient in practice once n is large enough. For small values of n the straightforward cubic algorithm is used instead. The crossover point where Strassen starts beating the cubic algorithm is determined empirically.
- Matrix multiplication has a long history, and there were many results that improved on Strassen's algorithm, including Strassen himself. Another well-known algorithm for matrix multiplication is due to Coppersmith and Winograd, 1978, and runs in $O(n^{2.376..})$. These algorithms are more efficient than Strassen for values of n so very very very very large (larger than the nb of particles in the universe) — they are only of practical interest.
- As of 2024, the best matrix multiplication algorithm runs in $O(n^{2.3727})$ by Virginia Williams, 2011.
- Improving matrix multiplication to $O(n^{2+\epsilon})$ is still an open problem! The lower bound is $\Omega(n^2)$ because there are n^2 elements in each matrix.