Divide-and-conquer

Module 4: Techniques

1 Overview

We have seen several problems so far whose solutions look very similar. This week we introduce a general 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.

2 Divide-and-conquer

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

Input: Problem P

Algorithm divideAndConquer(P):

1. Base case: if size of P is small, solve it (e.g. brute force) and return solution

   //else:

2. **Divide:** divide P into smaller problems P_1, P_2
3. **Conquer:** solve the smaller subproblems recursively, by calling divideAndConquer(P_1) and divideAndConquer(P_1)
4. **Combine:** combine solutions to P_1, P_2 into solution for P.

3 Example 1: Mergesort

- We've already seen a divide-and-conquer algorithm: It's Mergesort!

Mergesort an array of n elements:

- Base-case: if size of input is 1, return
- Else:
  - Divide: Divide the array into two arrays of $n/2$ elements each
  - Conquer: Sort the two arrays recursively
  - Combine: Merge the two sorted arrays of $n/2$ elements into one sorted array of size n

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

4 Example 2: Multiplying large integers

- The problem: We want to write an algorithm to multiply arbitrarily large numbers.
- Example: A = 13519384653184763746 and B = 32875641827561875665
- A and B cannot be represented as integers, because an integer (in any programming language) has a fixed precision. Basically in any programming language, an integer is represented on 4 bytes, which is 32 bits. Assuming the integer is unsigned, if all these bits are used for the value, the largest bvalue 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 in the computer is $4 \cdot 10^9$. If you want larger values you need to use special libraries for large numbers—which do what we'll do in this problem, represent values as arbitrarily long arrays of digits.
- So: we assume we are given two numbers, A and B, each one represented on n digits. We'll assume the digits are given as arrays:

$$A = [A_0, A_1, A_2...., A_{n-1}], B = [B_0, B_1, B_2..., B_{n-1}]$$

- We want to write an algorithm to compute the product of A and B: $C = A \times B$
- How do you compute C? Let's see an example.
- Example: $A = 357$, $B = 125$. We could do what we learnt in school, multiply each digit in A by each digit in B, and then add the results. Let's assume that the multiplication of two one-digit numbers takes $\Theta(1)$ time. When A and B have n digits each, this procedure we learnt in school would take $O(n^2)$ time. Right? Right.
- With our "algorithms hat" on, we ask the usual question: Can we do better than quadratic?

4.1 Towards a divide-and-conquer approach

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

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

- So we can say that $A \times B = (14 \cdot 10^2 + 27) \times (36 \cdot 10^2 + 59) = 14 \times 36 \cdot 10^4 + (27 \times 36 + 59 \times 14) \cdot 10^2 + 59 \times 27$. 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!

- Let's generalize: Let's denote by A' the first half of A and by $A"$ the second half of A, that is: $A = [A'A"]$, where $A' = [A_0, A_1, ...A_{n/2}]$ and $A" = [A_{n/2+1}, ..., A_{n-1}]$. We get that

$$A = A' \cdot 2^{n/2} + A"$$

- Remember we're working in base-2, everything is the same as in base 10, except with 2 instead of 10. For e.g. $1001_2 = (10 \cdot 2^2 + 01)_2 = (2 \times 2^2 + 1)_{10} = (2^3 + 1)_{10} = 9_{10}$

- Similarly, Let's denote by B' the first half of B and by $B"$ the second half of B, that is: $B = [B'B"]$, where $B' = [B_0, B_1, ...B_{n/2}]$ and $B" = [B_{n/2+1}, ..., B_{n-1}]$. We get that

$$B = B' \cdot 2^{n/2} + B"$$

- Now we can write

$$A \times B = (A' \cdot 2^{n/2} + A") \times (B' \cdot 2^{n/2} + B")$$

- Opening the parenthesis we get:

$$A \times B = A' \times B' \times 2^n + (A' \times B" + A" \times B') \times 2^{n/2} + A" \times B"$$

- What does this mean? To compute $A \times B$, the product of two n-digit numbers, we need to compute:

  1. We need to compute 4 products of two $n/2$-digit numbers: $A' \times B'$, $A' \times B"$, $A" \times B'$ and $A' \times B"$.
  2. We need to compute three sums of $\Theta(n)$-digit numbers (there are three "+" signs in the expression above). Adding 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 $A' \times B'$ by 2^n runs in the total number of bits in $A' \times B'$ plus n, which is $\Theta(n)$.
  4. Similarly, multiplying $(A' \times B" + A" \times B")$ by $2^{n/2}$ runs in $\Theta(n)$ time.

- So overall we expressed $A \times B$ as four products of $n/2$-digit numbers; once these products are known, it takes $\Theta(n)$ work to figure out $A \times B$.

- 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.
- Exercise: Solve the recurrence
- Really?
- Really. We get the same quadratic time as with the "straightforward" algorithm! Worse actually, because of recursion overhead.

4.2 Karatsuba's idea

- Still, the idea can be used to get a better algorithm.
- How? If we could express $A \times B$ 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 $\Theta(n^{lg3}) = n^{1.584} << n^2$
- But... How ??!
- Remember that

  $$A \times B = A' \times B' \cdot 2^n + (A' \times B" + A" \times B') \cdot 2^{n/2} + A" \times B"$$

  We need $A' \times B'$, $A' \times B" + A" \times B'$ and $A" \times B"$
- Karatsuba observed that we can compute $A' \times B" + A" \times B'$ in terms of the other two products but with some additional additions/subtractions:

  $A' \times B" + A" \times B' = (A' + A") \times (B' + B") - A' \times B' - A" \times B"$
- Therefore we would need only three products:

  $A' \times A", B' \times B"$ and $(A' + A") \times (B' + B")$
- The algorithm:

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

- We get the recurrence $T(n) = 3T(n/2) + \Theta(n)$, which solves to $\Theta(n^{lg3}) = n^{1.584}$
- Self-study exercise: Consider two numbers on 4 digits each, and compute their product using Karatsuba's algorithm; use base-10 for simplicity.

5 Example 3: Matrix Multiplication

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

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

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

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.

5.1 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.

  $X = \left\{ \begin{array}{cc} A & B \\ C & D \end{array} \right\}, Y = \left\{ \begin{array}{cc} E & F \\ G & H \end{array} \right\}$
- Their product $X \cdot Y$ can be written as:

  $\left\{ \begin{array}{cc} A & B \\ C & D \end{array} \right\} \cdot \left\{ \begin{array}{cc} E & F \\ G & H \end{array} \right\} = \left\{ \begin{array}{cc} (A \cdot E + B \cdot G) & (A \cdot F + B \cdot H) \\ (C \cdot E + D \cdot G) & (C \cdot F + D \cdot H) \end{array} \right\}$
- 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?

5.2 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:
  $$Z = \left\{ \begin{array}{cc} A & B \\ C & D \end{array} \right\} \cdot \left\{ \begin{array}{cc} E & F \\ G & H \end{array} \right\} = \left\{ \begin{array}{cc} (S_1 + S_2 - S_4 + S_6) & (S_4 + S_5) \\ (S_6 + S_7) & (S_2 + S_3 + S_5 - S_7) \end{array} \right\}$$
- 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

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

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

Some comments

- Strassen's algorithm "hides" a much bigger constant in $\Theta()$ than the straightfoward cubic algorithm.
- Currently the best known bound for matrix multiplication is $O(n^{2.376..})$ by Coppersmith and Winograd, 1978.
- The lower bound is (trivially) $\Omega(n^2)$.
- Improving matrix multiplication is still a big open problem!
- **In practice:** Strassen's algorithm 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 depends on the platform and needs to be determined empirically.
- **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 large values of n. For small values of n Karatsuba is fastest.