Algorithms: COMP3121/3821/9101/9801

School of Computer Science and Engineering
University of New South Wales Sydney

2. DIVIDE-AND-CONQUER

A Puzzle

- **An old puzzle:** We are given 27 coins of the same denomination; we know that one of them is counterfeit and that it is lighter than the others. Find the counterfeit coin by weighing coins on a pan balance only three times.
- **Solution:**
- This method is called "divide-and-conquer".
- We have already seen a prototypical "serious" algorithm designed using such a method: the MERGE-SORT.
- We split the array into two, sort the two parts recursively and then merge the two sorted arrays.
- We now look at a closely related but more interesting problem of counting inversions in an array.

Counting the number of inversions

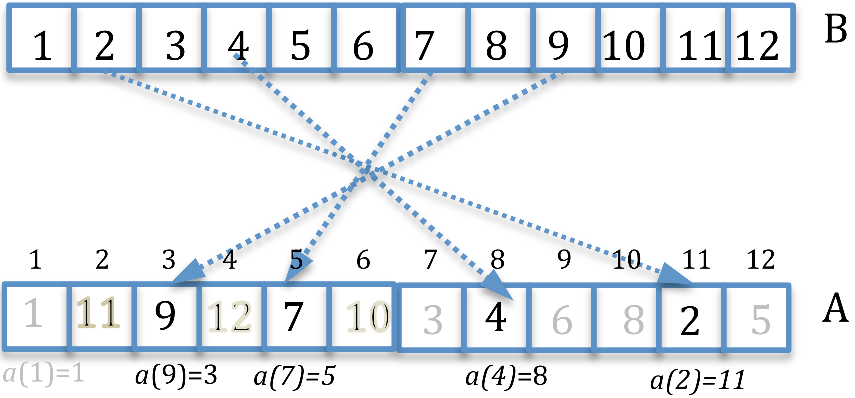
- Assume that you have m users ranking the same set of n movies. You want to determine for any two users A and B how similar their tastes are (for example, in order to make a recommender system).

- How should we measure the degree of similarity of two users A and B?

- Lets enumerate the movies on the ranking list of user B by assigning to the top choice of user B index 1, assign to his second choice index 2 and so on.

- For the i^{th} movie on $B's$ list we can now look at the position (i.e., index) of that movie on $A's$ list, denoted by $a(i)$.

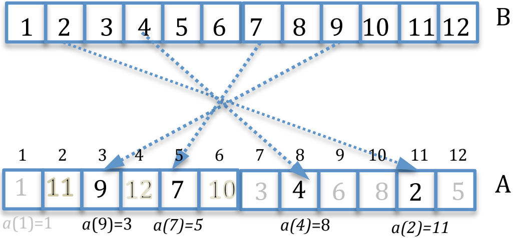
Counting the number of inversions



Counting the number of inversions

- A good measure of how different these two users are, is the total number of *inversions*, i.e., total number of pairs of movies i, j such that movie i precedes movie j on $B's$ list but movie j is higher up on $A's$ list than the movie i.

- In other words, we count the number of pairs of movies i, j such that $i < j$ (movie i precedes movie j on $B's$ list) but $a(i) > a(j)$ (movie i is in the position $a(i)$ on $A's$ list which is after the position $a(j)$ of movie j on $A's$ list.



- For example 1 and 2 do not form an inversion because $a(1) < a(2)$ ($a(1) = 1$ and $a(2) = 11$ because $a(1)$ is on the first and $a(2)$ is on the 11^{th} place in A);

- However, for example 4 and 7 do form an inversion because $a(7) < a(4)$ ($a(7) = 5$ because seven is on the fifth place in A and $a(4) = 8$)

Counting the number of inversions

- An easy way to count the total number of inversions between two lists is by looking at all pairs $i < j$ of movies on one list and determining if they are inverted in the second list, but this would produce a quadratic time algorithm, $T(n) = \Theta(n^2)$.

- We now show that this can be done in a much more efficient way, in time $O(n \log n)$, by applying a Divide-And-Conquer strategy.

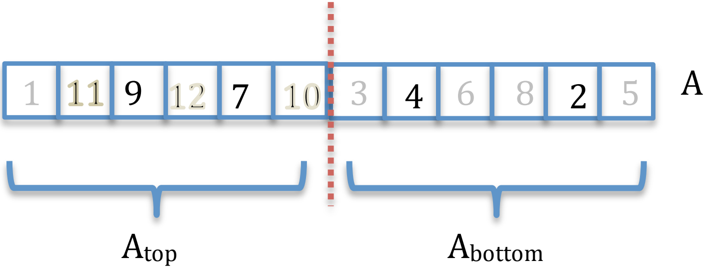
- Clearly, since the total number of pairs is quadratic in n, we cannot afford to inspect all possible pairs.

- The main idea is to tweak the Merge-Sort algorithm, by extending it to recursively both sort an array A **and** determine the number of inversions in A.

Counting the number of inversions

- We split the array A into two (approximately) equal parts $A_{top} = A[1 \ldots \lfloor n/2 \rfloor]$ and $A_{bottom} = A[\lfloor n/2 \rfloor + 1 \ldots n]$.

- Note that the total number of inversions in array A is equal to the sum of the number of inversions $I(A_{top})$ in A_{top} (such as 9 and 7) plus the number of inversions $I(A_{bottom})$ in A_{bottom} (such as 8 and 2) plus the number of inversions $I(A_{top}, A_{bottom})$ across the two halves (such as 7 and 4).

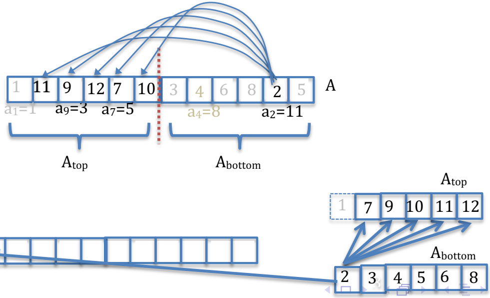


Counting the number of inversions

- We now recursively sort arrays A_{top} and A_{bottom} and obtain the number of inversions $I(A_{top})$ in the sub-array A_{top} and the number of inversions $I(A_{bottom})$ in the sub-array A_{bottom}.

- We now merge the two sorted arrays A_{top} and A_{bottom} while counting the number of inversions $I(A_{top}, A_{bottom})$ which are across the two sub-arrays.

- When the next smallest element among all elements in both arrays is an element in A_{bottom}, such an element clearly is in an inversion with all the remaining elements in A_{top} and we add the total number of elements remaining in A_{top} to the current value of the number of inversions across A_{top} and A_{bottom}.



Counting the number of inversions

- Whenever the next smallest element among all elements in both arrays is an element in A_{top}, such an element clearly is not involved in any inversions across the two arrays (such as 1, for example).
- After the merging operation is completed, we obtain the total number of inversions $I(A_{top}, A_{bottom})$ across A_{top} and A_{bottom}.
- The total number of inversions $I(A)$ in array A is finally obtained as:

$$I(A) = I(A_{top}) + I(A_{bottom}) + I(A_{top}, A_{bottom})$$

- **Next:** we study applications of divide and conquer to arithmetic of very large integers.

Basics revisited: how do we add two numbers?

```
 C C C C C     carry
   X X X X X    first integer
+  X X X X X    second integer
  -----------
 X X X X X X    result
```

- adding 3 bits can be done in constant time;
- the whole algorithm runs in linear time i.e., $O(n)$ many steps.

can we do it faster than in linear time?

- no, because we have to read every bit of the input
- no asymptotically faster algorithm

Basics revisited: how do we multiply two numbers?

```
      X X X X  <- first input integer
    * X X X X  <- second input integer
      -------
      X X X X  \
    X X X X     \ O(n^2) intermediate operations:
  X X X X       / O(n^2) elementary  multiplications
X X X X        /    + O(n^2) elementary additions
---------------
X X X X X X X X  <-  result of length 2n
```

- We assume that two `X`'s can be multiplied in $O(1)$. time (each `X` could be a bit or a digit in some other base).
- Thus the above procedure runs in time $O(n^2)$.
- Can we do it in **LINEAR** time, like addition?
- **No one knows!**
- "Simple" problems can actually turn out to be difficult!

Can we do multiplication faster than $O(n^2)$?

Let us try a divide-and-conquer algorithm:
take our two input numbers A and B, and split them into two halves:

$$A = A_1 2^{\frac{n}{2}} + A_0 \qquad \underbrace{XX\ldots X}_{\frac{n}{2}}\underbrace{XX\ldots X}_{\frac{n}{2}}$$

$$B = B_1 2^{\frac{n}{2}} + B_0$$

- $A_0,\ B_0$ - the least significant bits; $A_1,\ B_1$ the most significant bits.
- AB can now be calculated as follows:

$$AB = A_1 B_1 2^n + (A_1 B_0 + B_1 A_0) 2^{\frac{n}{2}} + A_0 B_0 \tag{1}$$

What we mean is that the product AB can be calculated recursively by the following program:

function $\textsc{Mult}(A, B)$
    if $|A| = |B| = 1$ **then return** AB
    else
        $A_1 \leftarrow \text{MoreSignificantPart}(A);$
        $A_0 \leftarrow \text{LessSignificantPart}(A);$
        $B_1 \leftarrow \text{MoreSignificantPart}(B);$
        $B_0 \leftarrow \text{LessSignificantPart}(B);$
        $X \leftarrow \textsc{Mult}(\mathrm{A_0}, \mathrm{B_0});$
        $Y \leftarrow \textsc{Mult}(\mathrm{A_0}, \mathrm{B_1});$
        $Z \leftarrow \textsc{Mult}(\mathrm{A_1}, \mathrm{B_0});$
        $W \leftarrow \textsc{Mult}(\mathrm{A_1}, \mathrm{B_1});$
        return $W\,2^n + (Y + Z)\,2^{n/2} + X$
    end if
end function

How many steps does this algorithm take?

Each multiplication of two n digit numbers is replaced by four multiplications of $n/2$ digit numbers: A_1B_1, A_1B_0, B_1A_0, A_0B_0, plus we have a **linear** overhead to shift and add:

$$T(n) = 4T\left(\frac{n}{2}\right) + c\,n \tag{2}$$

Can we do multiplication faster than $O(n^2)$?

Claim: if $T(n)$ satisfies

$$T(n) = 4T\left(\frac{n}{2}\right) + c\,n \tag{3}$$

then

$$T(n) = n^2(c+1) - c\,n$$

Proof: By "fast" induction. We assume it is true for $n/2$:

$$T\left(\frac{n}{2}\right) = \left(\frac{n}{2}\right)^2 (c+1) - c\,\frac{n}{2}$$

and prove that it is also true for n:

$$\begin{aligned} T\,(n) &= 4\,T\left(\tfrac{n}{2}\right) + c\,n = 4\left(\left(\tfrac{n}{2}\right)^2 (c+1) - \tfrac{n}{2}\,c\right) + c\,n \\ &= n^2(c+1) - 2c\,n + c\,n = n^2(c+1) - c\,n \end{aligned}$$

Can we do multiplication faster than $O(n^2)$?

Thus, if $T(n)$ satisfies $\quad T(n) = 4\,T\left(\frac{n}{2}\right) + c\,n \quad$ then

$$T(n) = n^2(c+1) - c\,n = O(n^2)$$

i.e., we gained **nothing** with our divide-and-conquer!

Is there a smarter multiplication algorithm taking less than $O(n^2)$ many steps??

Remarkably, there is, but first some history:
In 1952, one of the most famous mathematicians of the 20^{th} century, Andrey Kolmogorov, conjectured that you cannot multiply in less than $\Omega(n^2)$ elementary operations. In 1960, Karatsuba, then a 23-year-old student, found an algorithm (later it was called "divide and conquer") that multiplies two n-digit numbers in $\Theta(n^{\log_2 3}) \approx \Theta(n^{1.58\ldots})$ elementary steps, thus disproving the conjecture!! Kolmogorov was shocked!

The Karatsuba trick

How did Karatsuba do it??

Take again our two input numbers A and B, and split them into two halves:

$$A = A_1 2^{\frac{n}{2}} + A_0 \qquad \underbrace{XX\ldots X}_{\frac{n}{2}}\underbrace{XX\ldots X}_{\frac{n}{2}}$$

$$B = B_1 2^{\frac{n}{2}} + B_0$$

- AB can now be calculated as follows:

$$AB = A_1B_12^n + (A_1B_0 + A_0B_1)2^{\frac{n}{2}} + A_0B_0$$

$$= A_1B_12^n + ((A_1 + A_0)(B_1 + B_0) - A_1B_1 - A_0B_0)2^{\frac{n}{2}} + A_0B_0$$

- So we have saved one multiplication at each recursion round!

- Thus, the algorithm will look like this:

```
function MULT(A, B)
    if |A| = |B| = 1 then return AB
    else
        A_1 ← MoreSignificantPart(A);
        A_0 ← LessSignificantPart(A);
        B_1 ← MoreSignificantPart(B);
        B_0 ← LessSignificantPart(B);
        U ← A_0 + A_1;
        V ← B_0 + B_1;
        X ← MULT(A_0, B_0);
        W ← MULT(A_1, B_1);
        Y ← MULT(U, V);
        return W 2^n + (Y − X − W) 2^{n/2} + X
    end if
end function
```

- How fast is this algorithm?

The Karatsuba trick

Clearly, the run time $T(n)$ satisfies the recurrence

$$T(n) = 3\,T\left(\frac{n}{2}\right) + c\,n$$

and this implies (by replacing n with $n/2$)

$$T\left(\frac{n}{2}\right) = 3\,T\left(\frac{n}{2^2}\right) + c\,\frac{n}{2}$$

and by replacing n with $n/2^2$

$$T\left(\frac{n}{2^2}\right) = 3\,T\left(\frac{n}{2^3}\right) + c\,\frac{n}{2^2}$$

$$\dots$$

So we get $$T(n) = 3\,\underbrace{T\left(\frac{n}{2}\right)} + c\,n = 3\left(\underbrace{3T\left(\frac{n}{2^2}\right) + c\frac{n}{2}}\right) + c\,n$$

$$= 3^2\,\underbrace{T\left(\frac{n}{2^2}\right)} + c\frac{3n}{2} + c\,n = 3^2\left(\underbrace{3T\left(\frac{n}{2^3}\right) + c\frac{n}{2^2}}\right) + c\frac{3n}{2} + c\,n$$

$$= 3^3\,\underbrace{T\left(\frac{n}{2^3}\right)} + c\frac{3^2 n}{2^2} + c\frac{3n}{2} + c\,n = 3^3\left(\underbrace{3T\left(\frac{n}{2^4}\right) + c\frac{n}{2^3}}\right) + c\frac{3^2 n}{2^2} + c\frac{3n}{2} + c\,n = \dots$$

The Karatsuba trick

$$T(n) = 3T\left(\tfrac{n}{2}\right) + c\,n = 3\left(3T\left(\tfrac{n}{2^2}\right) + c\tfrac{n}{2}\right) + c\,n = 3^2\,\underbrace{T\left(\frac{n}{2^2}\right)} + c\tfrac{3n}{2} + c\,n$$

$$= 3^2\left(\underbrace{3T\left(\frac{n}{2^3}\right) + c\frac{n}{2^2}}\right) + c\tfrac{3n}{2} + c\,n = 3^3 T\left(\tfrac{n}{2^3}\right) + c\tfrac{3^2 n}{2^2} + c\tfrac{3n}{2} + c\,n$$

$$= 3^3\,\underbrace{T\left(\frac{n}{2^3}\right)} + c\,n\left(\tfrac{3^2}{2^2} + \tfrac{3}{2} + 1\right)$$

$$= 3^3\left(\underbrace{3T\left(\frac{n}{2^4}\right) + c\frac{n}{2^3}}\right) + c\,n\left(\tfrac{3^2}{2^2} + \tfrac{3}{2} + 1\right)$$

$$= 3^4 T\left(\tfrac{n}{2^4}\right) + c\,n\left(\tfrac{3^3}{2^3} + \tfrac{3^2}{2^2} + \tfrac{3}{2} + 1\right)$$

$$\ldots$$

$$= 3^{\lfloor \log_2 n \rfloor} T\left(\frac{n}{\lfloor 2^{\log_2 n}\rfloor}\right) + c\,n\left(\left(\tfrac{3}{2}\right)^{\lfloor \log_2 n \rfloor - 1} + \ldots + \tfrac{3^2}{2^2} + \tfrac{3}{2} + 1\right)$$

$$\approx 3^{\log_2 n} T(1) + c\,n\frac{\left(\frac{3}{2}\right)^{\log_2 n} - 1}{\frac{3}{2} - 1} = 3^{\log_2 n} T(1) + 2c\,n\left(\left(\tfrac{3}{2}\right)^{\log_2 n} - 1\right)$$

The Karatsuba trick

So we got

$$T(n) \approx 3^{\log_2 n} T(1) + 2c\,n\left(\left(\frac{3}{2}\right)^{\log_2 n} - 1\right)$$

We now use $a^{\log_b n} = n^{\log_b a}$ to get:

$$\begin{aligned} T(n) &\approx n^{\log_2 3} T(1) + 2c\,n\left(n^{\log_2 \frac{3}{2}} - 1\right) = n^{\log_2 3} T(1) + 2c\,n\left(n^{\log_2 3 - 1} - 1\right) \\ &= n^{\log_2 3} T(1) + 2c\,n^{\log_2 3} - 2c\,n \\ &= O(n^{\log_2 3}) = O(n^{1.58\ldots}) \ll n^2 \end{aligned}$$

Please review the basic properties of logarithms and the asymptotic notation from the review material (the first item at the class webpage under "class resources".)

A Karatsuba style trick also works for matrices: Strassen's algorithm for faster matrix multiplication

- If we want to multiply two $n \times n$ matrices P and Q, the product will be a matrix R also of size $n \times n$. To obtain each of n^2 entries in R we do n multiplications, so matrix product by brute force is $\Theta(n^3)$.
- However, we can do it faster using Divide-And-Conquer;
- We split each matrix into four blocks of (approximate) size $n/2 \times n/2$:

$$P = \begin{pmatrix} a & b \\ c & d \end{pmatrix}; \qquad Q = \begin{pmatrix} e & f \\ g & h \end{pmatrix}; \qquad R = \begin{pmatrix} r & s \\ t & u \end{pmatrix}.$$

- Then

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} \cdot \begin{pmatrix} e & f \\ g & h \end{pmatrix} = \begin{pmatrix} r & s \\ t & u \end{pmatrix} \tag{4}$$

A Karatsuba style trick also works for matrices: Strassen's algorithm for faster matrix multiplication

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} \cdot \begin{pmatrix} e & f \\ g & h \end{pmatrix} = \begin{pmatrix} r & s \\ t & u \end{pmatrix} \tag{5}$$

- We obtain:
$$\begin{aligned} a\,e + b\,g &= r & \qquad a\,f + b\,h &= s \\ c\,e + d\,g &= t & \qquad c\,f + d\,h &= u \end{aligned}$$

- Prima facie, there are 8 matrix multiplications, each running in time $T\left(\frac{n}{2}\right)$ and 4 matrix additions, each running in time $O(n^2)$, so such a direct calculation would result in time complexity governed by the recurrence

$$T(n) = 8T\left(\frac{n}{2}\right) + c\,n^2$$

- The first case of the Master Theorem gives $T(n) = \Theta(n^3)$, so nothing gained.

Strassen's algorithm for faster matrix multiplication

- However, we can instead evaluate:

$$A = a\,(f-h); \qquad B = (a+b)\,h; \qquad C = (c+d)\,e \qquad D = d\,(g-e);$$
$$E = (a+d)\,(e+h); \quad F = (b-d)(g+h); \quad H = (a-c)\,(e+f).$$

- We now obtain

$$\begin{aligned}
E + D - B + F &= (a\,e + d\,e + a\,h + d\,h) + (d\,g - d\,e) - (a\,h + b\,h) + (b\,g - d\,g + b\,h - d\,h)\\
&= a\,e + b\,g = r;\\
A + B &= (a\,f - a\,h) + (a\,h + b\,h) = a\,f + b\,h = s;\\
C + D &= (c\,e + d\,e) + (d\,g - d\,e) = c\,e + d\,g = t;\\
E + A - C - H &= (a\,e + d\,e + a\,h + d\,h) + (a\,f - a\,h) - (c\,e + d\,e) - (a\,e - c\,e + a\,f - c\,f)\\
&= c\,f + d\,h = u.
\end{aligned}$$

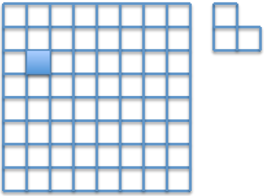
- We have obtained all 4 components of C using only 7 matrix multiplications and 18 matrix additions/subtractions.
- Thus, the run time of such recursive algorithm satisfies $T(n) = 7T(n/2) + O(n^2)$ and the Master Theorem yields $T(n) = \Theta(n^{\log_2 7}) = O(n^{2.808})$.
- In practice, this algorithm beats the ordinary matrix multiplication for $n > 32$.

Next time:

1. Can we multiply large integers faster than $O\left(n^{\log_2 3}\right)$??
2. Can we avoid messy computations like:

$$
\begin{aligned}
T(n) &= 3T\left(\frac{n}{2}\right) + cn = 3\left(3T\left(\frac{n}{2^2}\right) + c\frac{n}{2}\right) + cn = 3^2 T\left(\frac{n}{2^2}\right) + c\frac{3n}{2} + cn \\
&= 3^2\left(3T\left(\frac{n}{2^3}\right) + c\frac{n}{2^2}\right) + c\frac{3n}{2} + cn = 3^3 T\left(\frac{n}{2^3}\right) + c\frac{3^2 n}{2^2} + c\frac{3n}{2} + cn \\
&= 3^3 T\left(\frac{n}{2^3}\right) + cn\left(\frac{3^2}{2^2} + \frac{3}{2} + 1\right) = \\
&= 3^3\left(3T\left(\frac{n}{2^4}\right) + c\frac{n}{2^3}\right) + cn\left(\frac{3^2}{2^2} + \frac{3}{2} + 1\right) = \\
&= 3^4 T\left(\frac{n}{2^4}\right) + cn\left(\frac{3^3}{2^3} + \frac{3^2}{2^2} + \frac{3}{2} + 1\right) = \\
&\ldots \\
&= 3^{\lfloor \log_2 n \rfloor} T\left(\frac{n}{\lfloor 2^{\log_2 n} \rfloor}\right) + cn\left(\left(\frac{3}{2}\right)^{\lfloor \log_2 n \rfloor - 1} + \ldots + \frac{3^2}{2^2} + \frac{3}{2} + 1\right) \\
&\approx 3^{\log_2 n} T(1) + cn\frac{\left(\frac{3}{2}\right)^{\log_2 n} - 1}{\frac{3}{2} - 1} \\
&= 3^{\log_2 n} T(1) + 2cn\left(\left(\frac{3}{2}\right)^{\log_2 n} - 1\right)
\end{aligned}
$$

You are given a $2^n \times 2^n$ board with one of its cells missing (i.e., the board has a hole); the position of the missing cell can be arbitrary. You are also given a supply of "dominoes" each containing 3 such squares; see the figure:



Your task is to design an algorithm which covers the entire board with such "dominoes" except for the hole.

Hint: Do a divide-and-conquer recursion!

That's All, Folks!!