More Divide and Conquer + the Master Theorem

CMPUT 275 - Winter 2018

University of Alberta

Multiprecision Arithmetic

With `C++`, we seemed limited in the range of integers we could use. The biggest data type we saw used 64-bit integers.

But practical cryptography schemes like Diffie-Hellman and RSA use integer arithmetic where each integer has thousands of bits!

We can also use arbitrarily-large integers in Python.

How?
The onboard CPU only has circuits for performing arithmetic with bounded-size integers.

Multiprecision Arithmetic

Well, how could we even represent a large integer?

With an Array
Represent 63128418155489273 with an array `a[]` where:

$$\texttt{a[0]} = 3,\ \texttt{a[1]} = 7,\ \texttt{a[2]} = 2,\ \ldots,\ \texttt{a[16]}.$$

In practice, a different base is chosen (say, base 2^{32}) but the idea is the same.

Can keep an extra bit (or byte) of information indicating the sign (± 1) of the integer.

Addition

Addition and subtraction can be accomplished using the same algorithm you learned in elementary school.

```
  15342
+  5790
-------
= 21132
```

Say n is the number of digits in the larger number.
Running Time: $O(n)$.

Side Note: The number of digits in base $b \geq 2$ of a number N is around $\log_b N$. So our $O()$ running time bounds would be the same in a different (constant) base.

Multiplication?

We can multiply using the elementary-school algorithm as well.

```
     123
x    456
--------
     738
+   615
+  492
--------
=  56088
```

Again say n is the number of digits in the larger number.
Running Time: $O(n^2)$.

Faster Multiplication

We will see an algorithm with running time $O(n^{1.584963})$.

Where does this funny constant comes from? We will see.

The algorithm will use a **divide-and-conquer** scheme. But how to split the problem up?

Warmup
Let $a \cdot x + b$ and $c \cdot x + d$ be two polynomials over x (where a, b, c, d are values).

Note

$$(a \cdot x + b) \cdot (c \cdot x + d) = ac \cdot x^2 + (ad + bc) \cdot x + bd.$$

It takes four multiplications and one addition to compute the three coefficients $ac, ad + bc$, and bd.

A Puzzle
Figure out how to compute these three coefficients using only **three** multiplications and any constant number of additions/subtractions (my scheme uses 4 additions/subtractions).

Solution (Gauss' Trick)
Compute $R := a \cdot c, S := b \cdot d$ and $T := (a - b) \cdot (c - d)$.
Coefficient of x: $R + S - T$

This trick is the start of a divide-and-conquer algorithm!

Say we are multiplying to numbers N and M. Suppose N has the most digits (ignoring the sign) and that this is n.

Example: if $N = -14245$ we say there are $n = 5$ digits.

I will assume M also has n digits by padding it with zeros if necessary.

Example: if $N = 12345$ and $M = -678$ then I view M as -00678.

To compute $N \times M$, chop both numbers in half and recurse with 3 multiplications using numbers having $\approx n/2$ digits.

That is, write $|N|$ and $|M|$ as:

$$|N| = N_h \cdot 10^{n/2} + N_\ell$$

$$|M| = M_h \cdot 10^{n/2} + M_\ell$$

where $N_h, N_\ell, M_h, M_\ell < 10^{n/2}$ (suppose, for the moment, n is even).

How?
Simple: In our array representation of the digits of N and M, we get N_h from the top half of N's array, etc.

Example: if $N = 12345678$ then

$$N_h = 1234, \qquad N_\ell = 5678.$$

Recall

$$|N| = N_h \cdot 10^{n/2} + N_\ell$$
$$|M| = M_h \cdot 10^{n/2} + M_\ell$$

Note:

$$|N| \cdot |M| = A \cdot 10^n + B \cdot 10^{n/2} + C$$

where (using Gauss' trick):

$$A = N_h \cdot M_h, \qquad B = A - C - (N_h - N_\ell) \cdot (M_h - M_\ell), \qquad C = N_\ell \cdot M_\ell$$

This has us **recurse with three multiplications**, each with integers having **at most** $n/2$ **digits**.

Then compute $A \cdot 10^n$ by prepending n zeros and similarly for $B \cdot 10^{n/2}$.

Finally, add them together in $O(n)$ time and figure out the $\pm$ sign by looking at the sign of N and M.

Algorithm 1 `multiply`(N, M) each having at most n digits

```
if $n == 1$ then
    Return $N \cdot M$ by direct calculation.

if $n$ is odd then
    add a leading 0 to both $N$ and $M$ and increase $n$

get $N_h, N_\ell, M_h, M_\ell$ from the list representation of $|N|, |M|$

$A \leftarrow$ multiply$(N_h, M_h)$
$C \leftarrow$ multiply$(N_\ell, M_\ell)$
$B \leftarrow A - C -$ multiply$(N_h - N_\ell, M_h - M_\ell)$

return $(\pm 1) \cdot (A \cdot 10^n + B \cdot 10^{n/2} + C)$ # (whichever $\pm 1$ is appropriate)
```

Apart from the recursive calls, every other step runs in $O(n)$ time.

Running Time

A call with numbers with at most n digits uses $O(n)$ time in addition to subsequent recursive calls.

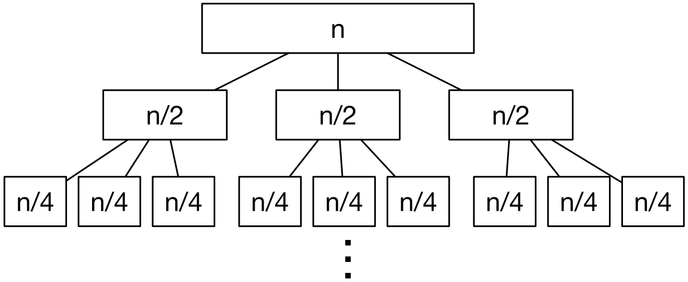
It recurses three times, each time with numbers having at most $\lceil n/2 \rceil$ digits (the smallest integer at least $n/2$).

So what is the final running time bound?

Let's analyze a slightly simpler case. What if each of the three recursive calls were guaranteed to have numbers with at most $n/2$ digits. **No ceilings**.

Running Time

Each box represents a call using numbers with the indicated size.



For $0 \leq i < \log_2 n$.
Number of recursive calls at depth i: 3^i

Running time within a recursive call at depth i: $n/2^i$.

Running Time

The running time is then bounded by some constant times:

$$\begin{aligned}
\sum_{i=0}^{\log_2 n} 3^i \cdot \frac{n}{2^i} &= n \cdot \sum_{i=0}^{\log_2 n} \left(\frac{3}{2}\right)^i \\
&= n \cdot \frac{(3/2)^{\log_2 n+1} - 1}{3/2 - 1} \\
&\leq 3 \cdot n \cdot n^{\log_2(3/2)} \\
&= 3 \cdot n^{\log_2 3}.
\end{aligned}$$

The idea is straightforward, but the sums are a bit messy. Generalizing to analyzing recursive calls with $\lceil n/2 \rceil$ digits seems nasty!

There is a systematic way to analyze these recurrences!

Master Theorem

The **Master Theorem** provides a simple answer to analyzing many divide-and-conquer schemes.

Let $T(n)$ be the running time bound for some recursive function on inputs of size n.

Suppose for some constants $a \geq 1, b > 1$ some function f that

$$T(n) = a \cdot T(\lceil n/b \rceil) + f(n).$$

In the case of the multiplication algorithm, $a = 3, b = 2$ and $f(n) \in O(n)$.

The Master Theorem allows us to express $T(n)$ as $O(g(n))$ for some simple function g. It comes in 3 cases.

Case 1: $f(n) = \Theta(n^c)$ where $c < \log_b a$

Recall $T(n) = a \cdot T(\lceil n/b \rceil) + f(n)$.

In this case, $T(n) = \Theta(n^{\log_b a})$.

If we ignored the $\lceil \cdot \rceil$, there will be a^i recursive calls at depth i, each taking $f(n/b^i) = \Theta((n/b^i)^c)$ time.

Total Work

$$\sum_{i=0}^{\log_b n} a^i \cdot \left(\frac{n}{b^i}\right)^c = n^c \sum_{i=0}^{\log_b n} \left(\frac{a}{b^c}\right)^i .$$

Summing and using rules of logarithms: the running time is $\Theta(n^{\log_b a})$.

Dealing with the ceilings is a technical nuisance, but the idea is the same.

Case 2: $f(n) = \Theta(n^c)$ where $c > \log_b a$

Recall $T(n) = a \cdot T(\lceil n/b \rceil) + f(n)$.

In this case, $T(n) = \Theta(n^c)$.

As before, there will be a^i recursive calls at depth i, each taking $f(n/b^i) = O((n/b^i)^c)$ time.

Total Work

$$\sum_{i=0}^{\log_b n} a^i \cdot \left(\frac{n}{b^i}\right)^c = n^c \sum_{i=0}^{\log_b n} \left(\frac{a}{b^c}\right)^i .$$

However, this time $a/b^c < 1$ so the sum is bounded by a constant. The work is, thus, $\Theta(n^c) = \Theta(f(n))$.

Case 3: $f(n) = \Theta(n^c \cdot \log^k n)$ where $c = \log_b a$

The most complicated case to remember, but we will see an examples of it with a real algorithm.

In this case, $T(n) = \Theta(n^c \cdot \log^{k+1} n)$.

Proof Idea
Since $c = \log_b a$ then $(a/b^c) = 1$. So the total work is

$$\sum_{i=0}^{\log_b n} a^i \cdot \left(\frac{n}{b^i}\right)^c \log^k(n/b^i) \leq n^c \log^k(n) \sum_{i=0}^{\log_b n} \left(\frac{a}{b^c}\right)^i.$$

Each term of the sum is 1, so it sums to $\log_b n$.

Can also show the $\geq$ direction to see $\Theta(n^c \cdot \log^{k+1} n)$ running time.

Multiplication

Recall the running time $T(n)$ of the multiplication algorithm satisfies:

$$T(n) = 3 \cdot T(n/2) + O(n).$$

Case 1 applies: as $f(n) = \Theta(n^1)$ and $1 < \log_b a = \log_2 3$.

So the running time is $\Theta(n^{\log_b a}) = \Theta(n^{\log_2 3})$ where $\log_2 3 \approx 1.584963$.

Parting Thoughts

This multiplication algorithm is called **Karatsuba's Algorithm** from the 1960s.

The fastest multiplication algorithm also uses divide-and-conquer and achieves running time $o(n \cdot \log n \cdot \log(\log n))$.

Division with remainder is much more complicated, but can be reduced to multiplication: given an algorithm for multiplication with running time $f(n)$ there is an $O(f(n))$ algorithm for division.

Cool Fact
The implementation of CPython (the interpreter we use) uses Karatsuba multiplication!
`https://github.com/python/cpython/blob/master/Objects/longobject.c`

Another Example: Strassen's Algorithm

The algorithm you learned in linear algebra to multiply two $n \times n$ square matrices runs uses $O(n^3)$ steps.

There is faster! Split the matrices into $(n/2) \times (n/2)$ matrices.

$$M_1 = \left[\begin{array}{c|c} A & B \\ \hline C & D \end{array}\right] \qquad M_2 = \left[\begin{array}{c|c} E & F \\ \hline G & H \end{array}\right]$$

Using a similar algebraic trick (look it up), can reduce to multiplying 7 pairs of $(n/2) \times (n/2)$ matrices and some $O(n^2)$ sums.

Running Time: $T(n) = 7 \cdot T(\lceil n/2 \rceil) + \Theta(n^2)$.
Master Theorem: $T(n) = \Theta(n^{\log_2 7})$.

More Master Theorem Practice

The Mergesort recurrence:

$$T(n) = a \cdot T(\lceil n/2 \rceil) + f(n)$$

where:

- $a = 2$
- $b = 2$
- $f(n) = \Theta(n)$

Thus $c = 1 = \log_2 2 = \log_b a$ so the tricky case of the Master Theorem applies:

$$T(n) = \Theta(n^c \cdot \log^{k+1} n)$$

In this case, $k = 0$ so we see $T(n) = \Theta(n \log n)$.

More Master Theorem Practice

It's a bit of overkill, but let's look at binary search.

$$T(n) = a \cdot T(\lceil n/2 \rceil) + f(n)$$

where:

- $a = 1$
- $b = 2$
- $f(n) = O(1)$

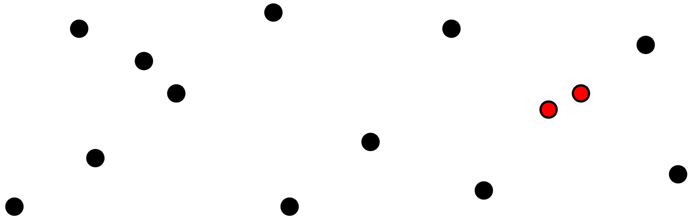
Thus $c = 0 = \log_2 1 = \log_b a$ so again the tricky case applies:

$$T(n) = O(n^c \cdot \log^{k+1} n)$$

Again in this case, $k = 0$ so we see $T(n) = O(\log n)$.

One Last Algorithm

Given: Points $p_1, \ldots, p_n$ in the plane where $p_i = (x_i, y_i)$.

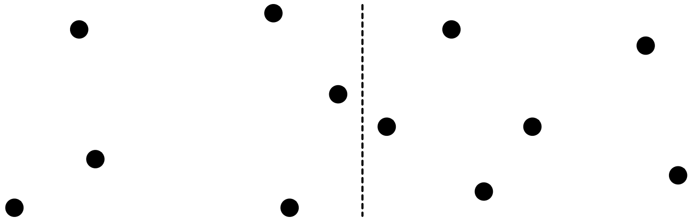


Find: The closest pair.

Obviously can do in $O(n^2)$ time, we will see better!

Divide-and-Conquer

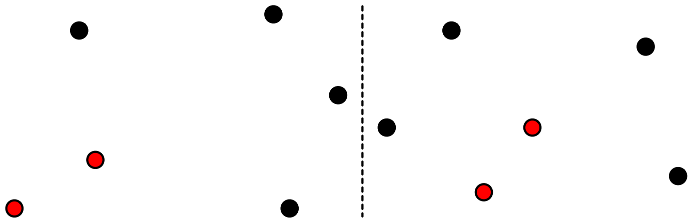
Sort the points by x-coordinate and split the list in two.



It is ok if many points have the same x-coordinate and the “cutting line” goes through many.

Divide-and-Conquer

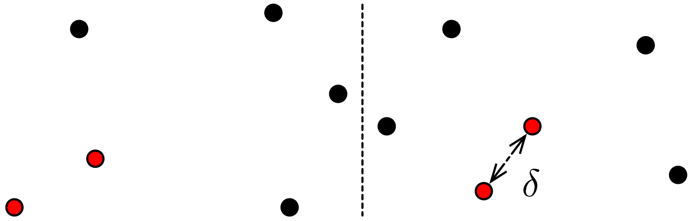
Recursion
Find the closest pair in each of the two sides!



Any closer pair must have one in the first half and one in the second.

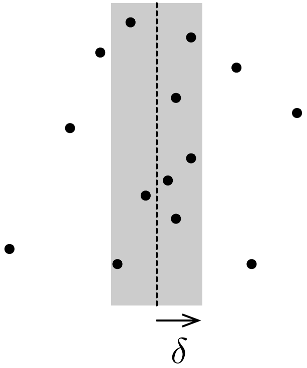
Pairs Between Sides

Let δ be the minimum distance pair on either side (i.e. the best solution found between both recursive calls).



Pairs Between Sides

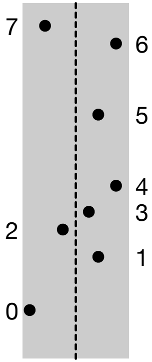
Look at the “strip” around the line separating the two sides of points. It has width δ on each side of the line.



If there is a closer pair, both endpoints lie in this strip.

Pairs Between Sides

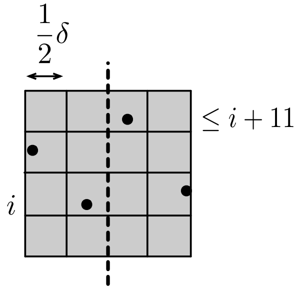
Sort all points (from both sides) on this strip by their *y*-coordinate.



For now, ignore points outside the strip.

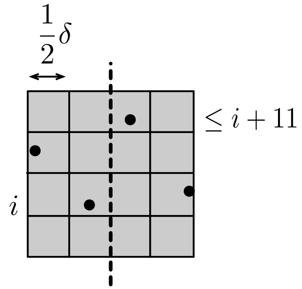
Pairs Between Sides

Claim: Any pair of points in this strip with distance $< \delta$ have their indices differ by at most 11.



To see this, break up the strip into a grid of square-length $\frac{\delta}{2}$ (can do without any point touching a horizontal grid line).

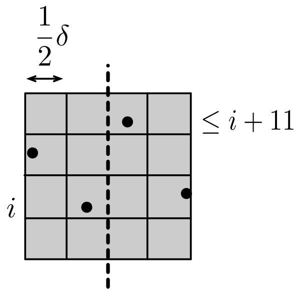
Pairs Between Sides



Assign points to the cell they are in. Any point on the middle line goes to the side it was assigned in the recursive calls.

If any two points lie in the same cell, they are on the same side and have distance $< \sqrt{(\delta/2)^2 + (\delta/2)^2} = \delta/\sqrt{2}$, contradicting the definition of δ. So each cell has at most one point.

Pairs Between Sides

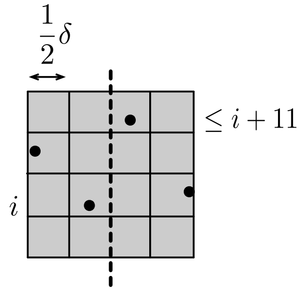


Finally, note if two points are distance $< \delta$ then there is at most one row of cells between them.

Each cell has at most one point, so their indices are at most 11 apart in the list of points in the strip (that are sorted by y-coordinates).

Pairs Between Sides

Summary, to find the closest pair of points with one point on each side, sort the points in the strip by their y-coordinate.



Compare each point i with each point j with $i < j \leq i + 11$ and keep the closest pair. If this is better than δ, return this pair.

Otherwise return the pair that had distance δ.

Algorithm 2 Closest pair of points in a set P with $n = |P|$.

```
if $n \leq 3$ then
    return the closest pair by trying all $\leq 3$ pairs
Sort the points $p_1, p_2, \ldots, p_n$ by $x$-coordinate
Recurse on $\{p_1, \ldots, p_{\lfloor n/2 \rfloor}\}$ and $\{p_{\lfloor n/2 \rfloor + 1}, \ldots, p_n\}$.
Let $\delta$ be the min. distance between both calls, with pair $C = \{p_i, p_j\}$.
Let $q_1, \ldots, q_k$ be the points of $P$ that are $\delta$-close to the line
        through $p_{\lfloor n/2 \rfloor}$, sorted by $y$ coordinate.
Find the closest pair $q_a, q_b$ among these points with $|a - b| \leq 11$.
return Either pair $C$ or $\{q_a, q_b\}$, whichever is closer.
```

Running Time: $T(n) = 2 \cdot T(\lceil n/2 \rceil) + O(n \log n)$.

The Master Theorem (tricky case again): $O(n \cdot \log^2 n)$.

Parting Thoughts

Can solve it in $O(n \log n)$ time. Only need to sort once by x-coordinate and once by y-coordinate.

A careful "merge" procedure of the points in the strip (like mergesort) can avoid sorting by y every recursive call.

Can also work harder and show the indices of the two closest two points with distance $< \delta$ (if any) differ by at most 6.

Can handle Manhattan distances, perhaps by tweaking the constant 11. Try it to see what suffices!