CS 341 - Algorithms, Spring 2017. University of Waterloo

Lecture 4: Divide and Conquer II

We will see more examples of divide and conquer algorithms: one in computational geometry and others in computer algebra, for which divide and conquer gives the fastest known algorithms.

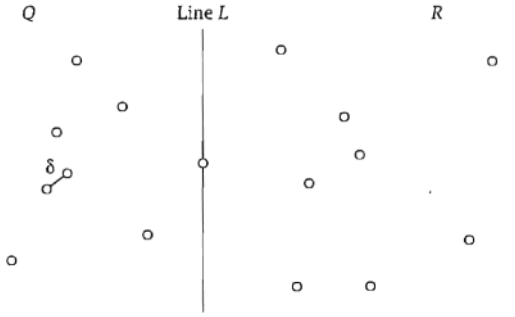
Closest Pair [KT 5.4]

Input: n points $(x_1,y_1), (x_2,y_2), \ldots, (x_n,y_n)$ on the 2D-plane.

Output: $1 \le i < j \le n$ that minimizes the Euclidean distance $\sqrt{(x_i-x_j)^2+(y_i-y_j)^2}$

It is clear that this problem can be solved in $O(n^2)$ time by trying all pairs.

We use the divide and conquer approach to give an improved algorithm.



picture from [KT 5.4]

We find a vertical line L to separate the point set into two halves: call the set of points on the left of the line Q, and the set of points on the right of the line R.

For simplicity, we assume that every point has a distinct x-value. We leave it as an exercise to see where this assumption is used and also how to remove it.

The vertical line can be found by computing the median based on the x-value, and put the first $\lceil \frac{n}{2} \rceil$ points in Q, and the last $\lfloor \frac{n}{2} \rfloor$ points in R.

Now, we recursively find the closest pair within Q, and the closest pair within R.

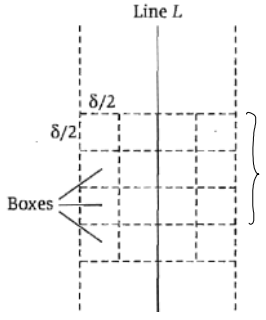
Suppose the closest pair in Q is of distance δ, and it is smaller than that in R.

To solve the closest pair problem in all n points, it remains to find the closest "crossing" pair with one point in Q and the other point in R.

It doesn't seem that the closest crossing pair problem is easier to solve.

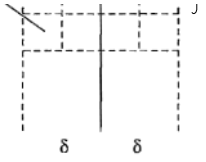
The idea is that we only need to determine whether there is a crossing pair with distance $< \delta$.

This allows us to restrict attention to the points with x-value within δ to the line L, but still all the points can be here.



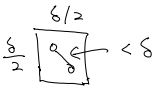
This allows us to restrict attention to the points with x-value within δ to the line L, but still all the points can be here.

We divide the narrow region into square boxes of side length $\frac{\delta}{2}$ as shown in the picture. Here comes the important observations.



(picture from [kT])

Observation 1 Each square box has at most one point.

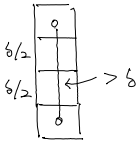


Proof If two points are in the same box, their distance is at most $\sqrt{\left(\frac{\delta}{2}\right)^2+\left(\frac{\delta}{2}\right)^2} = \frac{\delta}{\sqrt{2}} < \delta$.

This would contradict that the closest pairs within Q and within R have distance $\geq \delta$. □

Observation 2 Each point needs only to compute distances with points within two horizontal layers.

Proof For two points which are separated by at least two horizontal layers (see picture), then their distance would be more than δ and would not be closest. □



With observation 2, every point in a square box needs only to check with points in at most eleven other boxes (boxes in the same layer and the next two layers).

Observation 1 says that there is at most one point in each square.

Combining these, each point only needs to compute distances with at most eleven other points, in order to search for the closest pairs (i.e. pairs with distance $\leq \delta$).

This cuts down the search space from $\Omega(n^2)$ pairs to $O(n)$ pairs.

Algorithm

1. Find the dividing line L by computing the median using the x-value. Time: $O(n)$.
2. Recursively solve the closeset pair problem in Q and in R. Get δ. Time: $T(\frac{n}{2})$.
3. Using a linear scan, remove all the points not within the narrow region defined by δ. Time: $O(n)$.
4. Sort the points in non-decreasing order by their y-value. Time: $O(n \log n)$.
5. For each point, we compute its distance to the next eleven points in this y-ordering. Time: $O(n)$.
   (Note that two points within two layers must be within 11 points in the y-order, as ≤ 10 boxes in between.)
6. Return the minimum distance found.

The correctness of the algorithm is established by the two observations, justifying that it suffices for each point to compute distance to $O(1)$ other points as described in step 5.

Time complexity: $T(n) = 2T(\frac{n}{2}) + O(n \log n) \Rightarrow T(n) = O(n \log^2 n)$.

Note that the bottleneck is in the sorting step and it is not necessary to do sorting within recursion.

We can sort the points by y-value once in the beginning and use this ordering throughout the algorithm.

This reduces the time complexity to $T(n) = 2T(\frac{n}{2}) + O(n) \Rightarrow T(n) = O(n \log n)$.

Similarly, we don't need to compute the medians within the recursion.

We can sort the points by x-value once in the beginning and use it for the dividing step.

This would not improve the worst case time complexity but would improve its practical performance.

Questions: 1. Where did we use the assumption that the x-values are distinct?

2. What do we need to change so that the algorithm would work without this assumption?

Remark: There is a randomized algorithm to find a closest pair in expected $O(n)$ time. See [KT 13.7].

Arithmetic Problems

Arithmetic problems are where the divide and conquer approach is most powerful.

Many fastest algorithms for basic arithmetic problems are based on divide and conquer.

Today we will see some basic ideas how this approach works. But unfortunately we will not see the fastest algorithms as they require some background in algebra (although not too much).

Integer Multiplication [KT 5.6]

This is a really fundamental problem that our computers solve everyday.

Given two n-bit numbers $a = a_1 a_2 \cdots a_n$ and $b = b_1 b_2 \cdots b_n$, we would like to compute ab efficiently.

The multiplication algorithm that we learnt in elementary school takes $\Theta(n^2)$ bit operations.

```
   1100
 x 1101
------
   1100
  0000
 1100
1100
--------
10011100
```

} n n-bit numbers, adding one by one, each addition requires $\Theta(n)$ bit operations.

Let's apply the divide and conquer approach to integer multiplication.

Suppose we know how to multiply n-bit numbers efficiently.

We would like to use it to multiply 2n-bit numbers quickly.

Given two 2n-bit numbers x and y, we write $x = x_1 x_2$ and $y = y_1 y_2$, where x_1, y_1 are the higher-order n-bits and x_2, y_2 are the lower-order n-bits.

Written mathematically, $x = x_1 \cdot 2^n + x_2$ and $y = y_1 \cdot 2^n + y_2$.

Then, $xy = (x_1 \cdot 2^n + x_2)(y_1 \cdot 2^n + y_2) = x_1 y_1 \cdot 2^{2n} + (x_1 y_2 + x_2 y_1) \cdot 2^n + x_2 y_2$.

Since x_1, x_2, y_1, y_2 are n-bit numbers, the products x_1y_1, x_1y_2, x_2y_1, x_2y_2 can be computed recursively.

Therefore, $T(n) = 4T(\frac{n}{2}) + O(n)$, where the additional $O(n)$ bit operations are used to add the numbers.

(Note that $x_1y_1 \cdot 2^{2n}$ is simply shifting x_1y_1 to the left by $2n$ bits; we don't need a multiplication operation.)

Solving the recurrence will give $T(n) = O(n^2)$, not improving the elementary school algorithm.

This should not be surprising, since we haven't done anything clever to combine the subproblems, and we should not expect that just by doing divide and conquer some speedup would come automatically.

Karatsuba's algorithm: A clever way to combine the subproblems.

Instead of computing x_1y_1, x_1y_2, x_2y_1, x_2y_2 using four subproblems, Karatsuba's idea is to use three subproblems to compute x_1y_1, x_2y_2 and $(x_1+x_2)(y_1+y_2)$.

After that, we can compute the middle term $x_1y_2 + x_2y_1$ by noticing that

$$(x_1+x_2)\cdot(y_1+y_2) - x_1y_1 - x_2y_2 = x_1y_1 + x_1y_2 + x_2y_1 + x_2y_2 - x_1y_1 - x_2y_2 = x_1y_2 + x_2y_1.$$

That is, the middle term can be computed in $O(n)$ bit operations after solving the three subproblems.

Therefore, the total complexity is $T(n) = 3T(\frac{n}{2}) + O(n)$, and it follows from master theorem that $T(n) = O(n^{\log_2 3}) = O(n^{1.59})$.

This is the first and significant improvement over the elementary school algorithm.

Polynomial Multiplication

Given two degree n polynomials, $A(x) = a_n x^n + a_{n-1}x^{n-1} + \dots + a_1 x + a_0$ and $B(x) = \sum_{i=0}^{n} b_i x^i$.

we can use the same idea to compute $A \cdot B(x)$ in $O(n^{1.59})$ word operations,

where we assume that $a_i b_j$ can be computed in $O(1)$ word operations.

You will need to work out the details in the programming problem.

Matrix multiplication [DPV 2.5]

We all know the $O(n^3)$ word operations algorithm to multiply two nxn matrices.

The divide and conquer approach can also be successfully applied to matrix multiplication.

Given two 2nx2n matrices A and B, we can think of each matrix as consisting of four nxn blocks such that $A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}$ and $B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$.

Then, $$AB = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}\begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{pmatrix}$$

There are eight subproblems to solve: $A_{11}B_{11}$, $A_{11}B_{21}$, $A_{12}B_{21}$, $A_{12}B_{22}$, $A_{21}B_{11}$, $A_{21}B_{12}$, $A_{22}B_{21}$, $A_{22}B_{22}$.

After that, we just need to do $O(n^2)$ word operations to obtain AB by adding the subproblems.

The time complexity is $T(n) = 8T(\frac{n}{2}) + O(n^2)$, which implies that $O(n^3)$.

Again, this should not be surprising, as we are just doing the standard algorithm in block form.

We should not expect to get an improvement without doing anything clever.

Strassen's Algorithm

For some time, the $O(n^3)$ algorithm was thought to be optimal, but Strassen surprised the world by his magic formula: $AB = \begin{bmatrix} P_5 + P_4 - P_2 + P_6 & P_1 + P_2 \\ P_3 + P_4 & P_1 + P_5 - P_3 - P_7 \end{bmatrix}$,

where $P_1 = A_{11}(B_{12} - B_{22})$, $P_2 = (A_{11} + A_{12})B_{22}$, $P_3 = (A_{21} + A_{22})B_{11}$, $P_4 = A_{22}(B_{21} - B_{11})$,

$P_5 = (A_{11} + A_{22})(B_{11} + B_{22})$, $P_6 = (A_{12} - A_{22})(B_{21} + B_{22})$, $P_7 = (A_{11} - A_{21})(B_{11} + B_{12})$.

The point is that each subproblem can be computed using one multiplication and $O(n^2)$ additional operations.

The time complexity is $T(n) = 7T(\frac{n}{2}) + O(n^2)$.

It follows from the master theorem that $T(n) = O(n^{\log_2 7}) = O(n^{2.81})$!

After Strassen's algorithm, there is a long line of research (with some recent developments) pushing the time complexity of matrix multiplication to $O(n^{2.37})$.

Some researchers believe that matrix multiplication can be done in $O(n^2)$ word operations.

This is currently of theoretical interest only, as the algorithms are too complicated to be implemented.

Strassen's algorithm can be implemented and it will be faster than the standard algorithm when $n \gtrsim 5000$.

Applications

There are many combinatorial problems that can be reduced to matrix multiplication, and Strassen's result implies that they can be solved faster than $O(n^3)$ time.

As an example, the problem of determining whether a graph has a triangle can be reduced to matrix multiplication, and we leave it as a puzzle to you to figure out how.

There are many combinatorial problems in the literature where the fastest known algorithm is by matrix multiplication.

Fast Fourier Transform [DPV 2.6] [kT 5.7]

There is a very nice algorithm to solve integer multiplication and polynomial multiplication in $O(n \log n)$ time.

The presentation in [DPV 2.6] is highly recommended for those who are interested in learning it.

Waterloo has a strong symbolic computation group and you can learn it in CS 487.
