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.

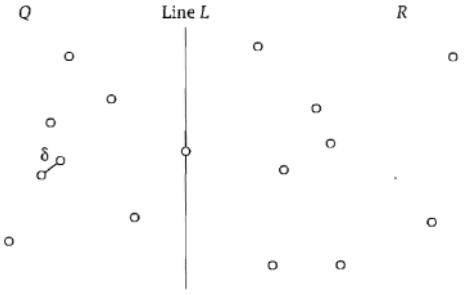
Closest pair [KT 5.4]

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.

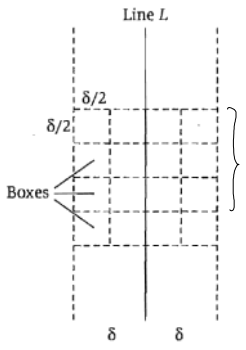
Suppose the closest pair in Q is of distance δ, and it is smaller than than in R.

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.

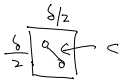


We divide this narrow region into square boxes of side length $\frac{\delta}{2}$ as

δ to the line L, but still all the points can be here.

(picture from [KT])

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



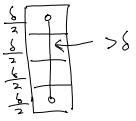
Observation 1 Each square box has at most one point.

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

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 horizonal layers (see picture), then their distance is more than δ and would not be optimal. □



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 only one point in each square box.

Combining them, each point only needs to compute distances with at most eleven other points (in order to search for the optimal pairs).

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

Algorithms

- Find the dividing line by computing the median using the x-value. ← $O(n)$.
- Recursively solve closest pair in Q, and closest pair in R. ← $2T(\frac{n}{2})$.
- Using a linear scan, we remove all points not within the narrow region. ← $O(n)$
- Then, we sort the points by their y-value in increasing order. ← $O(n\log n)$
- For each point, we just compute distances with the next eleven points in this ordering. ← $O(n)$
  (Two points within two layers must be within 11 points in the y-order, i.e. at most 10 points between them.)
- Return the minimum.

The above observations imply that the algorithm is correct.

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

The bottleneck is in the sorting step.

Note that we don't need to sort the y-value within the recursion.

We can sort the points by y-value and use it throughout the algorithm.

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

Similarly, we don't need to find medians within the recursion. We can keep another list with sorted x-value and use it in the "divide" step. This would not improve the time complexity but would improve the actual performance by a constant factor.

Questions - Where in the proofs did we use the assumption that the x-values are distinct?
- What do we need to change so that the algorithm will work without this assumption?

Remark: There is a randomized algorithm to find the 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 just show you some ideas how this approach works, but unfortunately we won't go into the faster algorithms since they also require some (not too much) background in algebra.

Integer multiplication [KT 5.6]

This is a really fundamental problem that our computers solve everyday.
Given two n-bit numbers $a = a_1 a_2 \dots a_n$ and $b = b_1 b_2 \dots b_n$, 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 number requires $\Theta(n)$ bit operations.

We apply the divide and conquer approach to integer multiplication.
Suppose we know how to multiply n-bit numbers quickly.
We would like to use it to multiply 2n-bit numbers quickly.
Given two 2n-bit numbers x and y, we write $x = x_1 x_0$ and $y = y_1 y_0$ where x_1, y_1 are the higher order n-bits and x_2, y_2 are the lower order n-bits.
Written mathematically, $x = x_1 2^n + x_0$ and $y = y_1 2^n + y_0$.
Then, $xy = (x_1 2^n + x_0)(y_1 2^n + y_0) = x_1 y_1 2^{2n} + (x_0 y_1 + x_1 y_0) 2^n + x_0 y_0$.
Since x_1, y_1, x_0, y_0 are n-bit numbers, the numbers $x_1 y_1$, $x_0 y_1$, $x_1 y_0$ and $x_0 y_0$ 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 $x_1y_1 2^{2n}$ is just shifting the number x_1y_1 to the left by $2n$ bits; we don't need a multiplication operation for it).

Solving the recurrence (say by master theorem) will give $T(n) = O(n^2)$, not improving the elementary school algorithm. This should not be surprising, since this is essentially the same algorithm as the elementary school one. We haven't done anything clever to combine the subproblems, and we shouldn't expect that just by divide and conquer some speedup would come automatically.

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

Instead of computing $x_1y_1, x_0y_1, x_1y_0, x_0y_0$ using four subproblems, the idea is to use three subproblems to compute x_1y_1, x_0y_0 and $(x_1+x_0)(y_1+y_0)$.

After that, we can recover the middle term $x_0y_1 + x_1y_0$ by noticing that

$$(x_1+x_0)(y_1+y_0) - x_1y_1 - x_0y_0 = x_1y_1 + x_0y_1 + x_1y_0 + x_0y_0 - x_1y_1 - x_0y_0 = x_0y_1 + x_1y_0.$$

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

Therefore, the total complexity is $T(n) = 3T(\frac{n}{2}) + O(n)$, and it follows from master's 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 $A(x) = a_n x^n + a_{n-1}x^{n-1} + \dots + a_1 x_1 + a_0$ and $B(x) = \sum_{i=0}^{n} b_i x^i$, we can use the same idea to compute $A \cdot B$ in $O(n^{1.59})$ word operations (where say a_ib_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 $n \times n$ matrices.

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

Given two $2n \times 2n$ matrices A and B, we can think of each matrix is consisted of four $n \times n$ 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, $$AB = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{pmatrix}$$

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

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 add them up to obtain AB.

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

Again, this should not be surprising, since we are just doing the standard algorithm in block form, and did not do anything clever.

For some time, it 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 P_i can be computed in one multiplication subproblem with additional $O(n^2)$ word operations. After computing all P_i, Strassen's formula tells us how to combine them to obtain AB in $O(n^2)$ word operations.

So, the time complexity is $T(n) = 7T(\frac{n}{2}) + O(n^2)$, and it follows from master's 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 recent developments) pushing the time complexity of matrix multiplication to $\approx O(n^{2.37})$.

This is currently only of theoretical interest, since the algorithm is so complicated that no one has implemented it.

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

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

Just to give you an idea, the problem of determining whether a graph has a triangle can be reduced to matrix multiplication (puzzle: how?), and thus Strassen's result implies that one can check whether a graph has a triangle in $O(n^{2.81})$ time, faster than trying all triples.

There are many problems in the literature where fastest algorithms are by matrix multiplication. We can discuss more when we have time.

Fast Fourier transform (optional) [KT 5.6] [DPV 2.6]

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

We just try to see the high-level idea, and I highly recommend the presentation in [DPV 2.6] for those who are interested in it.

Waterloo has a strong computer algebra group and you can learn it in CS 487.

Let's consider the polynomial multiplication problem.

The high-level idea is to change representations of the polynomials.

Instead of using the coefficient representation $A(x) = \sum_{i=0}^{n} a_i x^i$ and $B(x) = \sum_{i=0}^{n} b_i x^i$, the idea is to represent a polynomial by its evaluation at $n+1$ points.

That is, we find $n+1$ distinct points $p_1, \ldots, p_{n+1}$ and store $A(p_1), \ldots, A(p_{n+1})$.

We know that a polynomial can be recovered uniquely from its evaluation in $n+1$ points (e.g. a line can be recovered uniquely from 2 points, etc).

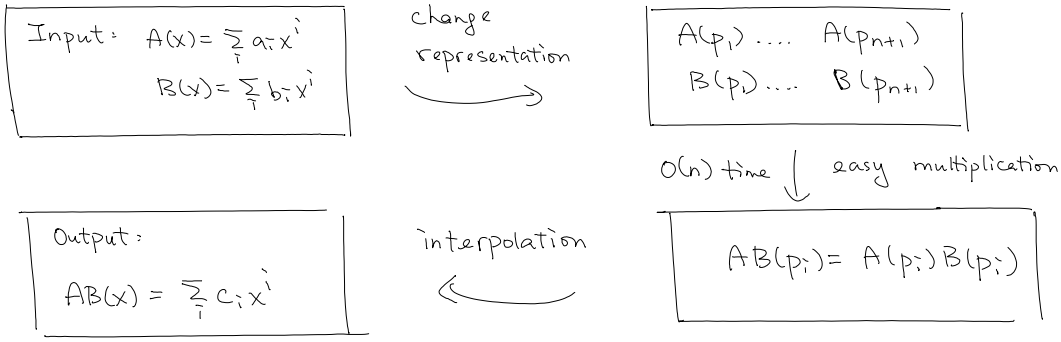
Why are we considering this representation?

The key point is that once we have $A(p_1), \ldots, A(p_{n+1})$ and $B(p_1), \ldots, B(p_{n+1})$, then we can compute $AB(p_1), \ldots, AB(p_{n+1})$ easily, because $AB(p_i)$ is just $A(p_i) \cdot B(p_i)$.

So, in this representation, polynomial multiplication can be done in $O(n)$ word operations (assuming the values $A(p_i)$, $B(p_i)$ are reasonably bounded).

Now, if we also have fast algorithms to change representations back and forth, then we would have a fast algorithm for polynomial multiplication.

The scheme is as follows:



We have to come up with fast algorithms for the first step and the last step.

For the first step, the key is to pick $n+1$ good points, so that there will be a lot of symmetry so that computing them together will be much faster than

$n+1$ independent evaluations since much work can be shared by symmetry.

It turns out the good points to use are $p_i = \omega^i$ where ω is the $(n+1)$-th root of unity (complex number), and the first step can be done in $O(n \log n)$ time.

The best way to see it is through linear algebra, since computing $A(1), A(\omega), A(\omega^2), \ldots, A(\omega^n)$ can be viewed as

$$\begin{bmatrix} A(1) \\ A(\omega) \\ A(\omega^2) \\ \vdots \\ A(\omega^n) \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega & \omega^2 & \cdots & \omega^{n-1} \\ 1 & \omega^2 & \omega^4 & \cdots & \omega^{2n-2} \\ \vdots & \vdots & & & \\ 1 & \omega^{(n-1)} & & & \omega^{(n-1)(n-1)} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{bmatrix}.$$

The matrix has a lot of symmetry and the matrix-vector multiplication can be done efficiently by a divide and conquer algorithm.

Interestingly, once we cast the problem in this linear algebraic form, the final interpolation step can also be written in this form, and it turns out that the last step can also be done in $O(n \log n)$ time.

This efficient way of changing the representation is called the fast Fourier transform, and it has many applications in signal processing and even some in combinatorics.

For more details, please read [DPU 2.6].
