Algorithms

CS 341

Lap Chi Lau

LATEXed by Sibelius Peng

Preface

Disclaimer Much of the information on this set of notes is transcribed directly/indirectly from the lectures of CS 341 during Spring 2021 as well as other related resources. I do not make any warranties about the completeness, reliability and accuracy of this set of notes. Use at your own risk.

I couldn't create a good set of notes from Winter 2020 (the term I was in), so I decided to create a set of notes based on Prof. Lau's version. The notes is solely based on his written course notes, not his videos, not his PPTs. Because I am familiar with the concepts, I might skip lots of details in the notes. Please use this set of notes at your own risk.

The main focus of the course is on the design and analysis of efficient algorithms, and these are fundamental building blocks in the development of computer science. We will cover

- divide and conquer and solving recurrence,
- simple graph algorithms using BFS and DFS,
- greedy algorithms,
- dynamic programming,
- bipartite matching,
- NP-completeness and reductions.

The idea of reduction can also be found in CS 365 and CS 360.

There are three reference books and we refer to them using the following shorthands:

- [DPV] Algorithms, by Dasgupta, Papadimitriou and Vazirani.
- [KT] Algorithm design, by Kleinberg and Tardos.
- [CLRS] Introduction to algorithms, by Cormen, Leiserson, Rivest, and Stein.

Note that Ronald **R**ivest is one of the creators of RSA cryptosystem. See CO 487 for more details.

For any questions, send me an email via https://notes.sibeliusp.com/contact.

You can find my notes for other courses on https://notes.sibeliusp.com/.

Sibelius Peng

Contents

1 Introduction

To introduce you to the course, different instructors use different examples. Stinson uses 3SUM problem. During Fall 2019, Lubiw & Blais (and possible for several future offerings) develop algorithms for merging two convex hulls, which is quite interesting. However, in Winter 2020, the motivating example was max subarray problem, which was studied already in CS 136, maybe not in CS 146. Now let's dive into Spring 2021 offering.

1.1 Two Classical Problems

We are given an undirected graph with n vertices and m edges, where each edge has a non-negative cost. The **traveling salesman problem** asks us to find a minimum cost tour to visit every vertex of the graph at least once (visit all cities). The **Chinese postman problem** asks us to find a minimum cost tour to visit every edge of the graph at least once (visit all streets).

A naive algorithm to solve the TSP is to enumerate all permutations and return the minimum cost one. This takes $O(n!)$ time which is way too slow. By using dynamic programming, we can solve it in $O(2^n)$ time, and this is essentially the best known algorithm that we know of.

We will prove this problem is "NP-complete", and probably efficient algorithms for this problem do not exist. However, people may design some approximation algorithms, which will be covered in CS 466 and CO 754.

Surprisingly, the Chinese postman problem, which looks very similar, can be solved in $O(n^4)$, using techniques from graph matching.

1.2 Time Complexity

How do we define the time complexity of an algorithm?

Roughly speaking, we count the number of operations that the algorithm requires. One may count exactly how many operations. The precise constant is probably machine-dependent and may be also difficult to work out. So, the standard practice is to use asymptotic time complexity to analyze algorithms.

Asymptotic Time Complexity

Given two functions $f(n), g(n)$, we say

- (upper bound, big-O) $g(n) = O(f(n))$ if $\lim_{n\to\infty} \frac{g(n)}{f(n)} \le c$ for some constant c (independent of n).
- (lower bound, big-Ω) $g(n) = \Omega(f(n))$ if $\lim_{n\to\infty} \frac{g(n)}{f(n)} \ge c$ for some constant c.
- (same order, big-Θ) $g(n) = \Theta(f(n))$ if $\lim_{n\to\infty} \frac{g(n)}{f(n)} = c$ for some constant c.
- (loose upper bound, small-o) $g(n) = o(f(n))$ if $\lim_{n\to\infty} \frac{g(n)}{f(n)} = 0$.
- (loose lower bound, small-ω) $g(n) = \omega(f(n))$ if $\lim_{n\to\infty} \frac{g(n)}{f(n)} = \infty$.

Worst Case Complexity

We say an algorithm has time complexity $O(f(n))$ if it requires at most $O(f(n))$ primitive operations for all inputs of size n (e.g., n bits, n numbers, n vertices, etc). By adopting the asymptotic time complexity, we are ignoring the leading constant and lower order terms.

"Good" Algorithms

For most optimization problems, such as TSP, there's a straightforward exponential time algorithm. For those problems, we are interested in designing a polytime algorithm for it.

1.3 Computation models

Note that this section often appears in the short answer questions from in-person midterms, in order to trick the student...

When we say that we have an $O(n^4)$-time algorithm, we need to be more precise about what are the primitive operations that we assume. In this course, we usually assume the **word-RAM** model, in which we assume that we can access an arbitrary position of an array in constant time, and also that each word operation (such as addition, read/write) can be done in constant time. This model is usually good in practice, because the problem size can usually be fit in the main memory, and so the word size is large enough and each memory access can be done in more or less the same time.

For problems like computing the determinant, we usually consider the bit-complexity, i.e., how many bit operations involved in the algorithm. So please pay some attention to these assumptions when we analyze the time complexity of an algorithm, especially for numerical problems. That said, the word-complexity and bit-complexity usually don't make a big difference in this course (e.g., at most a $\log(n)$ factor) and so we just use the word-RAM model.

However, in some past midterms, the questions might trick you on this. Also, it does make a difference when we analyze an algorithm in terms of the input size. Read the trial division example in Section 7.4.1 of CO 487.

1.4 3SUM

3SUM

We are given $n+1$ numbers $a_1, a_2, \ldots, a_n$ and c and we would like to determine if there are i, j, k such that $a_i + a_j + a_k = c$.

Algorithm 1 Enumerate all triples and check whether its sum is c. Time: $O(n^3)$.

Algorithm 2 Observe that $a_i + a_j + a_k = c$ can be rewritten as $c - a_i - a_j = a_k$. We can enumerate all paris $a_i + a_j$ and check whether $c - a_i - a_j$ is equal to some a_k.

To do this efficiently, we can first sort the n numbers so that $a_1 \leq a_2 \leq \cdots \leq a_n$. Then checking whether $c - a_i - a_j = a_k$ for some k via binary search in $O(\log n)$. So the total complexity is

$$O(\underset{\substack{\uparrow \\ \text{sorting}}}{n \log n} + \underset{\substack{\uparrow \\ \text{binary search} \\ \text{for each pair}}}{n^2 \log n}) = O(n^2 \log n).$$

Algorithm 3 We can rewrite the condition: $a_i + a_j = c - a_k$. Suppose again that we sort n numbers so that $a_1 \leq a_2 \leq \cdots \leq a_n$. The idea is that given this sorted array and the number $b := c - a_k$, we can check whether there are i, j such that $a_i + a_j = b$. In other words, the 2-SUM problem can be solved in $O(n)$ time given the sorted array. If this is true, we then get an $O(n^2)$-time algorithm for 3-SUM, by trying $b := c - a_k$ for each $1 \leq k \leq n$. That is, we reduce the 3-SUM problem to n instances of the 2-SUM problem.

Algorithm 1: 2-SUM

```
$L := 1, R := n$ // left index and right index
while $L \leq R$ do
    if $a_L + a_R = c$ then
        DONE
    else if $a_L + a_R > c$ then
        $R \leftarrow R - 1$
    else
        $L \leftarrow L + 1$
```

Proof of correctness If there are no i, j such that $a_i + a_j = b$, then we won't find them. Now suppose $a_i + a_j = b$ for some $i \leq j$. Since the algorithm runs until $L = R$, there is an iteration such that $L = i$ or $R = j$. WLOG, suppose that $L = i$ happens first, and $R > j$; the other case is symmetric. As long as $R > j$, we have $a_i + a_R > a_i + a_j = b$, and so we will decrease R until $R = j$, and so the algorithm will find this pair. □

Time Complexity $O(n)$m because $R - L$ decrease by one in each iteration, so the algorithm will stop within $n - 1$ iterations.

2

Divide and Conquer

2.1 Merge Sort

Sorting is a fundamental algorithmic task, and merge sort is a classical algorithm using the idea of divide and conquer. This divide and conquer approach works if there is a nice way to reduce an instance of the problem to smaller instances of the same problem. The merge sort algorithm can be summarized as follows:

Algorithm 2: Merge sort

```
Function Sort($A[1,n]$):
    if $n = 1$ then
        return
    Sort($A\left[1, \lceil \frac{n}{2} \rceil\right]$)
    Sort($A\left[\lceil \frac{n}{2} \rceil + 1, n\right]$)
    merge$\left(A\left[1, \lceil \frac{n}{2} \rceil\right], A\left[\lceil \frac{n}{2} \rceil + 1, n\right]\right)$
```

The correctness of the algorithm can be proved formally by a standard induction. We focus on analyzing the running time. We can draw a recursion tree as shown in the course note. Then the asymptotic complexity of merge-sort is $O(n \log n)$. This complexity can also be proved by induction, by using the "guess and check" method.

2.2 Solving Recurrence

By drawing the recursion tree and discussing by cases, we can derive the master theorem:

Master Theorem

If $T(n) = aT\left(\frac{n}{b}\right) + n^c$ for constants $a > 0, b > 1, c \geq 0$, then

$$T(n) = \begin{cases} O(n^c) & \text{if } c > \log_b a \\ O(n^c \log n) & \text{if } c = \log_b a \\ O(n^{\log_b a}) & \text{if } c < \log_b a \end{cases}$$

Single subproblem

This is common in algorithm analysis. For example,

- $T(n) = T\left(\frac{n}{2}\right) + 1$, we have $T(n) = O(\log n)$, binary search.
- $T(n) = T\left(\frac{n}{2}\right) + n$, we have $T(n) = O(n)$, geometric sequence.
- $T(n) = T(\sqrt{n}) + 1$, we have $T(n) = O(\log \log n)$, counting levels.

  (In level i, the subproblem is of size $n^{2^{-i}}$. When $i = \log \log n$, it becomes $n^{\frac{1}{\log n}} = O(1)$.)

Non-even subproblems

We will see one interesting example later.

- $T(n) = T\left(\frac{2n}{3}\right) + T\left(\frac{n}{3}\right) + n$. We have $T(n) = O(n \log n)$.
- $T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{4}\right) + n$. We have $T(n) = O(n)$.

Exponential time

$T(n) = 2T(n-1) + 1$. We have $T(n) = O(2^n)$. Can we improve the runtime if we have $T(n) = T(n-1) + T(n-2) + 1$? This is the same recurrence as the Fibonacci sequence. Using "computing roots of polynomials" from MATH 249, it can be shown that

$$T(n) = O\left(\left(\frac{1+\sqrt{5}}{2}\right)^n\right) = O(1.618^n),$$

faster exponential time.

Consider the maximum independent set problem:

Maximum independent set

Given $G = (V, E)$. We want to find a maximum subset of vertices $S \subseteq V$ suc that there are no edges between every pair of vertices $u, v \in S$.

A naive algorithm is to enumerate all subsets, taking $\Omega(2^n)$ time. Now consider a simple variant.

Pick a vertex v with maximum degree.

- If $v \notin S$, delete v and reduce the graph size by one.
- If $v \in S$, we choose v, and then we know that all neighbors of v cannot be chosen, and so we can delete v and all its neighbors, so that the graph size is reduced by at least two.

So $T(n) \leq T(n-1) + T(n-2) + O(n)$, and it is strictly smaller than $O(2^n \cdot n)$.

2.3 Counting Inversions

Input: n distinct numbers, $a_1, a_2, \ldots, a_n$
Output: number of paris with $i < j$ but $a_i > a_j$

For example, given $(3, 7, 2, 5, 4)$, there are five pairs of inversions $(3, 2)$, $(7, 2)$, $(7, 5)$, $(7, 4)$, $(5, 4)$.

We can think of this problem as computing the "unsortedness" of a sequence. We may also imagine that this is measuring how different are two rankings.

For simplicity, we again assume that $n = 2^k$ for k integer. Using the idea of divide and conquer, we try to break the problem into two halves. Suppose could count the number of inversions in the first half, as well as in the second half. Would it then be easier to solve the remaining problem?

$$\underbrace{\begin{array}{|c|c|c|c|}\hline 3 & 9 & 2 & 4 \\ \hline\end{array}}_{\text{3 pairs}}\underbrace{\begin{array}{|c|c|c|c|}\hline 6 & 1 & 7 & 8 \\ \hline\end{array}}_{\text{1 pair}}$$

It remains to count the number of inversions with one number in the first half and the other number in the second half. These "cross" inversion pairs ar easier to count, because we know their relative positions. In particular, to facilitate the counting, we could sort the first half and the second half, without worrying losing information as we already counted the inversion pairs with each half.

$$\begin{array}{|c|c|c|c|}\hline 2 & 3 & 4 & 9 \\ \hline\end{array} \qquad \begin{array}{|c|c|c|c|}\hline 1 & 6 & 7 & 8 \\ \hline\end{array}$$

Now, for each number c in the second half, the number of "cross" inversion pairs involving c is precisely the number of numbers in the first half that is larger than c. In this example, 1 is involved in 4 cross pairs, $6, 7, 8$ are all involved in 1 cross pair (with 9), and so the number of "cross" inversion pairs is 7.

How to count the number of cross inversion pairs involving a number a_j in the second half efficiently?

Idea 1 as the first half is sorted, we can use binary search to determine how many numbers in the first half are greater than a_j. This takes $O(\log n)$ time for one a_j, and totally $O(n \log n)$ time for all numbers in the second half. This is not too slow, but we can do better.

Idea 2 Observe that this information can be determined when we merge the two sorted list in merge sort. When we insert a number in the second half to the merged list, we know how many numbers in the first half that are greater than it.

For example,

$$\begin{array}{|c|c|c|c|}\hline 2 & 3 & 4 & 9 \\ \hline \multicolumn{1}{c}{} & \multicolumn{1}{c}{} & \multicolumn{1}{c}{} & \multicolumn{1}{c}{\uparrow}\end{array} \qquad \begin{array}{|c|c|c|c|}\hline 1 & 6 & 7 & 8 \\ \hline \multicolumn{1}{c}{} & \multicolumn{1}{c}{\uparrow} & \multicolumn{1}{c}{} & \multicolumn{1}{c}{}\end{array} \quad \Longrightarrow \quad \begin{array}{|c|c|c|c|c|c|c|c|}\hline 1 & 2 & 3 & 4 & 6 & & & \\ \hline\end{array}$$

we know that there is only one number in the first half that is greater than 6. As in merge sort, this can be done in $O(n)$ time.

Algorithm 3: Count cross inversions

```
Function count($A[1, n]$):
    if $n = 1$ then return 0
    count($A\left[1, \frac{n}{2}\right]$)
    count($A\left[\frac{n}{2} + 1, n\right]$)
    merge-and-count-cross-inversions$\left(A\left[1, \frac{n}{2}\right], A\left[\frac{n}{2} + 1, n\right]\right)$
```

Total time complexity is $T(n) = 2T\left(\frac{n}{2}\right) + O(n \log n)$. Solving this will give $T(n) = \Theta(n \log^2 n)$.

The sorting step is the bottleneck and unnecessary as we have sorted them in the merge step. As it turns out, we can just modify the merge sort algorithm to count the number of inversion pairs.

Algorithm 4: Count cross inversions (final version)

```
Function count-and-sort($A[1, n]$):
    if $n = 1$ then return $0$
    $s_1 \leftarrow$ count-and-sort($A\left[1, \frac{n}{2}\right]$)
    $s_2 \leftarrow$ count-and-sort($A\left[\frac{n}{2}+1, n\right]$)
    $s_3 \leftarrow$ merge-and-count-cross-inversions$\left(A\left[1, \frac{n}{2}\right], A\left[\frac{n}{2}+1, n\right]\right)$
    return $s_1 + s_2 + s_3$
```

Total time complexity $T(n) = 2T\left(\frac{n}{2}\right) + O(n)$. Solving this will give us $T(n) = O(n \log n)$.

2.4 Maximum Subarray

See CLRS 4.1. Skipped. $O(n)$ solution is using dynamic programming.

2.5 Finding Median

Input: n distinct numbers $a_1, a_2, \ldots, a_n$.
Output: the median of these numbers.

It's clear thar problem can be solved in $O(n \log n)$ time by first sorting the numbers, but it turns out that there is an interesting $O(n)$-time algorithm. To solve the median problem, it is more convenient to consider a slightly more general problem.

Input: n distinct numbers $a_1, a_2, \ldots, a_n$ and an integer $k \geq 1$.
Output: the k-th smallest number in $a_1, \ldots, a_n$.

The reason is that the median problem doesn't reduce to itself (and so we can't recurse), while the k-th smallest number lends itself to reduction as we will see.

The idea is similar to that in quicksort (which is a divide and conquer algorithm). We choose a number a_i. Split the n numbers into two groups, one group with numbers smaller than a_i, called it S_1, and the other group with numbers greater than a_i, called it S_2.

Let r be the rank of a_i, i.e., a_i is the r-th smallest number in $a_1, \ldots, a_n$.

- If $r = k$, then we are done.
- If $r > k$, the nwe find the k-th smallest number in S_1.
- If $r < k$, then find the $(k - r)$-th smallest number in S_2.

Observe that when $r > k$, the problem size is reduced to $r - 1$ as $|S_1| = r - 1$, and when $r < k$, the problem size is reduced to $n - r$ as $|S_2| = n - r$. So if somehow we could choose a number "in the middle" as a pivot as in quicksort, then we can reduce the problem size quickly and making good progress, but finding a number in the middle is the very question that we want to solve. But obverse that we don't need the pivot to be exactly in the middle, just that it is not too close to the boundary.

Suppose we can choose a_i such that its rank satisfies say $\frac{n}{10} \leq r \leq \frac{9n}{10}$, then we know that the problem size would have reduced by at least $\frac{n}{10}$, as $|S_1| = r - 1 \leq \frac{9n}{10}$ and $|S_2| = n - r \leq \frac{9n}{10}$. So, the recurrence relation for the time complexity is $T(n) \leq \left(\frac{9n}{10}\right) + P(n) + cn$, where $P(n)$ denotes the time to find a good pivot and cn is the number of operations for splitting. if we manage to find a good pivot point in $O(n)$ time, i.e., $P(n) = O(n)$, the it implies that $T(n) = O(n)$. We have made some progress to the median problem, by reducing the problem of finding the number exactly in the middle to the easier problems of finding a number not too far from the middle.

It remains to figure out a linear time algorithm to find a good pivot.

Randomized solution If we have seen randomized quicksort before, then we would have guessed that keep choosing a random pivot would work. This is indeed the case and we can take a look at [DPV 2.4] for a proof. Details not included in this course, but in CS 761.

Deterministic solution There is an interesting deterministic algorithm that would always return a number a_i with rank $\frac{3n}{10} \leq r \leq \frac{7n}{10}$ in $O(n)$ time. As seen above, this means $P(n) = O(n)$ and it follows that $T(n) = O(n)$.

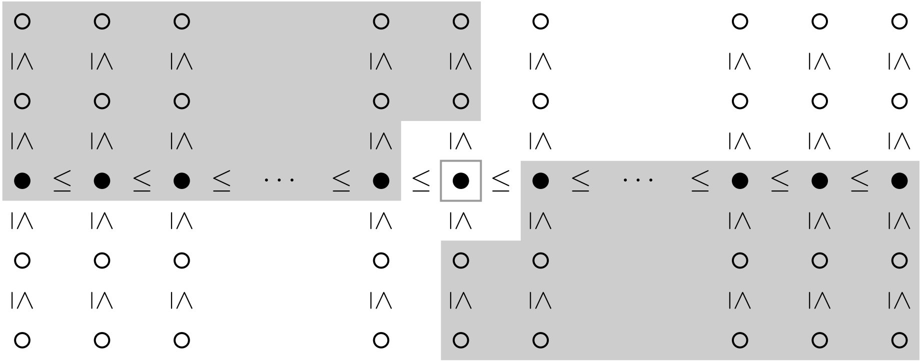
Finding a good pivot The idea of the algorithm is to find the median of medians.

1. Divide the n numbers into $\frac{n}{5}$ groups, each of 5 numbers. Time: $O(n)$.
2. Find the median in each group. Call them $b_1, b_2, \ldots, b_{\frac{n}{5}}$. Time: $O(n)$.
3. Find the median of these $\frac{n}{5}$ medians $b_1, b_2, \ldots, b_{\frac{n}{5}}$. Time: $O(n)$.

Lemma

Let r be the rank of the median of medians. Then $\frac{3n}{10} \leq r \leq \frac{7n}{10}$.

Proof:



The square is the median of the medians.

In the picture, we sort each group, and then order the groups by an increasing order of the medians. We emphasize that this is just for the analysis, and we don't need to do sorting in the algorithm. It should be clear that the square is greater than the numbers in the top-left corner, and is smaller than the numbers in the bottom-right corner.

There are about $3 \cdot \frac{n}{10} = \frac{3n}{10}$ numbers in the top-left and bottom-right corners. This implies that $\frac{3n}{10} \leq 4 \leq \frac{7n}{10}$. □

This lemma proves the correctness of the pivoting algorithm.

Time complexity

We have $P(n) = T\left(\frac{n}{5}\right) + c_1 n$, where c_1 is a constant. By the reduction above,

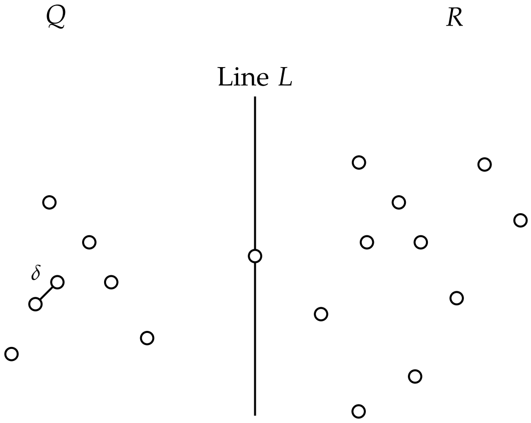
$$T(n) \leq T\left(\frac{7n}{10}\right) + P(n) + c_2 n = T\left(\frac{7n}{10}\right) + T\left(\frac{n}{5}\right) + (c_1 + c_2)n$$

Then $T(n) = O(n)$.

2.6 Closest Pair

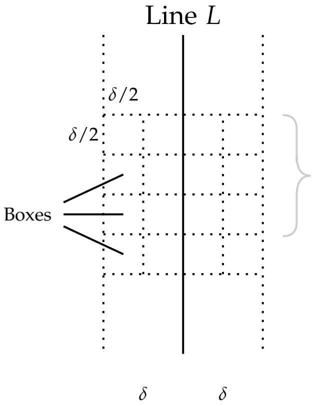
Input: n points $(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$ on the 2D-plane.
Output: $1 \le i < j \le n$ that minimizes the Euclidean distance $\sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$

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 given an improved algorithm.



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.

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



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 closes pairs within Q and within R have distance $\ge \delta$. □

Observaion 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, then their distance would be more than δ and would not be closest. □

With observation 2, every 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 $\Omega(n^2)$ to $O(n)$ pairs.

Algorithm 5: Finding the minimum distance

```
1 Find the dividing line $L$ by computing the median using the $x$-value.
2 Recursively solve the closest pair problem in $Q$ and in $R$. Get $\delta$.
3 Using a linear scan, remove all the points not within the narrow region defined by $\delta$.
4 Sort the points in non-decreasing order by their $y$-value.
5 For each point, we compute its distance to the next eleven points in this $y$-ordering. // Note
      that two points within two layers must be within 11 points in the $y$-order, as
      $\leq 10$ 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 $T(n) = 2T\left(\frac{n}{2}\right) + O(n \log n)$. Solving this gives us $T(n) = O(n \log^2 n)$.

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 $T(n) = 2T\left(\frac{n}{2}\right) + O(n) \Rightarrow T(n) = O(n \log n)$.

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?

There is a randomized algorithm to find a closest pair in expected $O(n)$ time.

2.7 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 sme background in algebra.

Integer Multiplication

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 learnt in elementary school takes $\Theta(n^2)$ 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 apply it to $2n$-bit numbers.

Given two $2n$-bit numbers x and y, we write $x = x_1x_2$ and $y = y_1y_2$, where x_1, y_1 are higher-order n-bits and x_2, y_2 are the lower-order n-bits. In other words, $x = x_1 \cdot 2^n + x_2$ and $y = y_1 \cdot 2^n + y_2$. Then

$$xy = (x_1 \cdot 2^n + x_2)(y_1 \cdot 2^n + y_2) = x_1y_1 \cdot 2^{2n} + (x_1y_2 + x_2y_1) \cdot 2^n + x_2y_2 \,.$$

Since x_1, x_2, y_1, y_2 are n-bit numbers, then the products here can be computed recursively. Therefore, $T(n) = 4(\frac{n}{2}) + O(n)$, 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.

Consider **Karatsuba's algorithm**, which is a clever way to combine the subproblems. Instead of computing 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, $A(x) = \sum_{i=0}^{n} a_i x^i$ 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.

Matrix multiplication

We have already known $O(n^3)$ *word* operations algorithm to multiply two $n \times n$ matrices. If we use divide and conquer, namely divide a matrix into 4 blocks, and get 8 subproblems. Then we still get $O(n^3)$ complexity because we haven't done anything clever.

See Strassen's algorithm for $O(n^{2.81})$ complexity. After that, $O(n^{2.37})$ was achieved. Some researchers believe that it 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 $n \gtrsim 5000$.

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 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 algorithm is by matrix multiplication.

Faster Fourier Transform

This is a very nice algorithm to solve integer multiplication and polynomial multiplication in $O(n \log n)$ time. See [DPV 2.6] or CS 371.