CSS.201.1 Algorithms

Instructor: Umang Bhaskar

TIFR 2024, Aug-Dec

Scribe: Soham Chatterjee

sohamchatterjee999@gmail.com
Website: sohamch08.github.io

Contents

Chapter 1

Finding Closest Pair of Points

Find-Closest(S)
Input: Set $S = \{(x_i, y_i) \mid x_i, y_i \in \mathbb{R},\ \forall\ i \in [n]\}$. We denote $P_i = (x_i, y_i)$.
Question: Given a set of points find the closest pair of points in $\mathbb{R}^2$ find P_i, P_j that are at minimum l_2 distance i.e. minimize $\sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$.

1.1 Naive Algorithm

Now the naive algorithm for this will be checking all pairs of points and take their distance and output the minimum one. There are total $\binom{n}{2}$ possible choices of pairs of points. And calculating the distance of each pair takes $O(1)$ time. So it will take $O(n^2)$ times to find the closest pair of points.

Idea: $\forall\ P_i, P_j \in S$ find distance $d(P_i, P_j)$ and return the minimum. Time taken is $O(n^2)$.

1.2 Divide and Conquer Algorithm

Below we will show a Divide and Conquer algorithm which gives a much faster algorithm.

Definition 1.2.1: Divide and Conquer

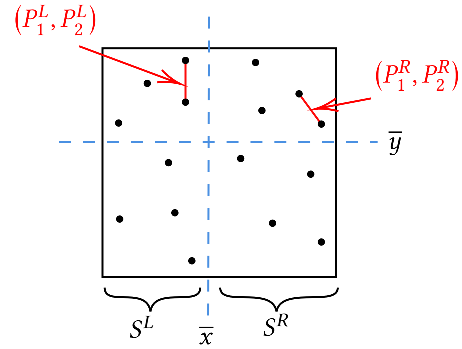
- Divide: Divide the problem into two parts (roughly equal)
- Conquer: Solve each part individually recursively. If the subproblem sizes are small enough, however, just solve the subproblems in a straightforward manner.
- Combine: Combine the solutions to the subproblems into the solution.

1.2.1 Divide

So to divide the problem into two roughly equal parts we need to divide the points into two equal sets. That we can do by sorting the points by their $x-$coordinate. Suppose S^x denote we get the new sorted array or points. And similarly we obtain S^y which denotes the array of points after sorting S by their $y-$coordinate.

Algorithm 1: Step 1 (Divide)

Function *Divide*:
- Sort S by $x-$coordinate and $y-$coordinate
- $S^x \longleftarrow S$ sorted by $x-$coordinate
- $S^y \longleftarrow S$ sorted by $y-$coordinate
- $\bar{x} \longleftarrow \lfloor \frac{n}{2} \rfloor$ highest $x-$coordinate
- $\bar{y} \longleftarrow \lfloor \frac{n}{2} \rfloor$ highest $y-$coordinate
- $S^L \longleftarrow \{P_i \mid x_i < \bar{x},\ \forall\ i \in [n]\}$
- $S^R \longleftarrow \{P_i \mid x_i \geq \bar{x},\ \forall\ i \in [n]\}$



1.2.2 Conquer

Now we will recursively get pair of closest points in S_L and S_R. Suppose the (P_1^L, P_2^L) are the closest pair of points in S^L and (P_1^R, P_2^R) are the closest pair of points in S^R.

Algorithm 2: Step 1 (Solve Subproblems)

Function *Conquer*:
- Solve for S_L, S^R.
- (P_1^L, P_2^L) are closest pair of points in S_L.
- (P_1^R, P_2^R) are closest pair of points in S_R.
- $\delta^L = d(P_1^L, P_2^L)$, $\delta^R = d(P_1^R, P_2^R)$
- $\delta_{min} \longleftarrow \min\{\delta^L, \delta^R\}$

1.2.3 Combine

Now we want to combine these two solutions.

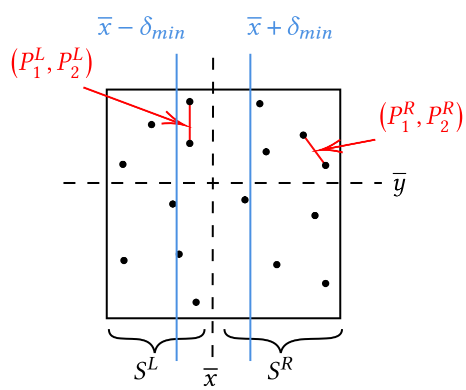
Question 1.1: We are not done

Is there a pair of points $P_i, P_j \in S$ such that $d(P_i, P_j) < \delta_{min}$

If Yes:

- One of them must be in S_L and the other is in S_R.
- $x-$coordinate $\in [\overline{x} - \delta_{min}, \overline{x} + \delta_{min}]$.
- $|y_i - y_j| \leq \delta_{min}$

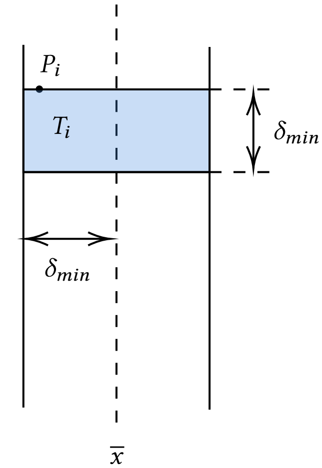
So we take the strip of radius δ_{min} around $\overline{x}$. Define $T = \{P_i \in S \mid |x_i - \overline{x}| \leq \delta_{min}\}$



We now sort all the points in the T by their decreasing $y-$coordinate. Let T_y be the array of points. For each $P_i \in T_y$ define the region

$$T_i = \{P_j \in T_y \mid 0 \leq y_j - y_i \leq \delta_{min}, j > i\}$$

> **Lemma 1.2.1**
>
> Number of points (other than P_i) that lie inside the box is at most 8



Proof: Suppose there are more than 8 points that lie inside the box apart from P_i. The box has a left square part and a right square part. So one of the squares contains at least 5 points. WLOG suppose the left square has at least 5 points. Divide each square into 4 parts by a middle vertical and a middle horizontal line. Now since there are 5 points there is one part which contains 2 points but that is not possible as those two points are in S_L and their distance will be less than δ_{min} which is not possible. Hence contradiction. Therefore there are at most 8 points inside the box. ■

Hence by the above lemma for each $P_i \in T_y$ there are at most 8 points in T_i. So for each $P_j \in T_i$ we find the $d(P_i, P_j)$ and if it is less than δ_{min} we update the points and the distance

1.2.4 Pseudocode and Time Complexity

Assumption. *We will assume for now that for all $P_i.P_j \in S$ we have $x_i \neq x_j$ and $y_i \neq y_j$. Later we will modify the pseudocode to remove this assumption*

Algorithm 3: FIND-CLOSEST(S)

Input: Set of n points, $S = \{(x_i, y_i) \mid x_i, y_i \in \mathbb{R},\ \forall\, i \in [n]\}$. We denote $P_i = (x_i, y_i)$.
Output: Closest pair of ponts, (P_i, P_j, δ) where $\delta = d(P_i, P_j)$

1 **begin**
2   **if** $|S| \leq 10$ **then**
3   Solve by Brute Force (Consider every pair of points)
4   $S^x \longleftarrow S$ sorted by $x-$coordinate, $S^y \longleftarrow S$ sorted by $y-$coordinate
5   $\overline{x} \longleftarrow \lfloor \frac{n}{2} \rfloor$ highest $x-$coordinate, $\overline{y} \longleftarrow \lfloor \frac{n}{2} \rfloor$ highest $y-$coordinate
6   $S^L \longleftarrow \{P_i \mid x_i < \bar{x},\ \forall\, i \in [n]\}$, $S^R \longleftarrow \{P_i \mid x_i \geq \bar{x},\ \forall\, i \in [n]\}$
7   $(P_1^L, P_2^L, \delta^L) \longleftarrow$ FIND-CLOSEST(S^L), $(P_1^R, P_2^R, \delta^R) \longleftarrow$ FIND-CLOSEST(S^R)
8   $\delta_{min} \longleftarrow \min\{\delta^L, \delta^R\}$
9   **if** $\delta_{min} < \delta^L$ **then**
10     $P_1 \longleftarrow P_1^R$, $P_2 \longleftarrow P_2^R$
11   **else**
12     $P_1 \longleftarrow P_1^L$, $P_2 \longleftarrow P_2^L$
13   $T \longleftarrow \{P_i \mid |x_i - \overline{x}| \leq \delta_{min}\}$
14   $T_y \longleftarrow T$ sorted by decreasing $y-$coordinate
15   **for** $P \in T_y$ **do**
16     $U \longleftarrow$ Next 8 points
17   **for** $\hat{P} \in U$ **do**
18   **if** $d(P, \hat{P}) < \delta_{min}$ **then**
19         $\delta_{min} \longleftarrow d(P, \hat{P})$
20         $(P_1, P_2) \longleftarrow (P, \hat{P})$
21   **return** (P_1, P_2, δ_{min})

Notice we used the assumption in the line 5 for finding the medians. So the line 4 takes $O(n \log n)$ times. Lines 5,6 takes $O(n)$ time. Since $\overline{x}$ is the median, we have $|S^L| = \lfloor \frac{n}{2} \rfloor$ and $|S^R| = \lceil \frac{n}{2} \rceil$. Hence FIND-CLOSEST($S^L$) and FIND-CLOSEST($S^R$) takes $T\left(\frac{n}{2}\right)$ time. Now lines $8 - 12$ takes constant time. Line 13 takes $O(n)$ time. And line 14 takes $O(n \log n)$ time. Since U has 8 points i.e. constant number of points the lines $16 - 20$ takes constant time for each $P \in T_y$. Hence the for loop at line 15 takes $O(n)$ time. Hence total time taken

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

1.3 Improved Algorithm for $O(n \log n)$ Runtime

Notice once we sort the points by $x-$coordinate and $y-$coordinate we don't need to sort the points anymore. We can just pass the sorted array of points into the arguments for solving the smaller problems. Their is another time where we need to sort which is in line 14 of the above algorithm. This we can get actually from S^y without sorting just checking one by one backwards direction if the $x-$coordinate of the points satisfy $|x_i - \overline{x}| \leq \delta_{min}$. So

$$T_y = \textsc{Reverse}(\{P_i \in S^y \mid |x_i - \overline{x}| \leq \delta_{min}\})$$

So we form a new algorithm which takes the input S^x and S^y and then finds the closest pair of points. Then we will use that subroutine to find closest pair of points in any given set of points.

Algorithm 4: $\textsc{Find-Closest-Sorted}(S^x, S^y)$

Input: Set of n points, $S = \{(x_i, y_i) \mid x_i, y_i \in \mathbb{R},\ \forall\, i \in [n]\}$.
S^x and S^y are the sorted array of points with respect to $x-$coordinate and $y-$coordinate respectively

Output: Closest pair of ponts, (P_i, P_j, δ) where $\delta = d(P_i, P_j)$

begin
  if $|S| \leq 10$ **then**
    Solve by Brute Force
  $\overline{x} \longleftarrow \lfloor \frac{n}{2} \rfloor$ highest $x-$coordinate
  $\overline{y} \longleftarrow \lfloor \frac{n}{2} \rfloor$ highest $y-$coordinate
  $S^L \longleftarrow \{P_i \in S^x \mid x_i < \bar{x},\ \forall\, i \in [n]\}$
  $S^L_y \longleftarrow \{P_i \in S^y \mid x_i < \overline{x}\}$
  $S^R \longleftarrow \{P_i \in S^x \mid x_i \geq \bar{x},\ \forall\, i \in [n]\}$
  $S^R_y \longleftarrow \{P_i \in S^y \mid x_i \geq \overline{x}\}$
  $(P^L_1, P^L_2, \delta^L) \longleftarrow \textsc{Find-Closest-Sorted}(S^L, S^L_y)$
  $(P^R_1, P^R_2, \delta^R) \longleftarrow \textsc{Find-Closest-Sorted}(S^R, S^R_y)$
  $\delta_{min} \longleftarrow \min\{\delta^L, \delta^R\}$
  if $\delta_{min} < \delta^L$ **then**
    $P_1 \longleftarrow P^R_1, P_2 \longleftarrow P^R_2$
  else
    $P_1 \longleftarrow P^L_1, P_2 \longleftarrow P^L_2$
  $T \longleftarrow \{P_i \mid |x_i - \overline{x}| \leq \delta_{min}\}$
  $T_y \longleftarrow \textsc{Reverse}(\{P_i \in S^y \mid |x_i - \overline{x}| \leq \delta_{min}\})$
  for $P \in T_y$ **do**
    $U \longleftarrow$ Next 8 points
    for $\hat{P} \in U$ **do**
      if $d(P, \hat{P}) < \delta_{min}$ **then**
        $\delta_{min} \longleftarrow d(P, \hat{P})$
        $(P_1, P_2) \longleftarrow (P, \hat{P})$
  return (P_1, P_2, δ_{min})

Algorithm 5: $\textsc{Find-Closest}(S)$

Input: Set of n points,
$S = \{(x_i, y_i) \mid x_i, y_i \in \mathbb{R},\ \forall\, i \in [n]\}$.
We denote $P_i = (x_i, y_i)$.

Output: Closest pair of ponts, (P_i, P_j, δ) where $\delta = d(P_i, P_j)$

begin
  if $|S| \leq 10$ **then**
    Solve by Brute Force
  $S^x \longleftarrow S$ sorted by $x-$coordinate
  $S^y \longleftarrow S$ sorted by $y-$coordinate
  return *Find-Closest-Sorted*(S^x, S^y)

This algorithm only sorts one time. So time complexity for $\textsc{Find-Closest-Sorted}(S^x, S^y)$ is

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

and therefore times complexity for $\textsc{Find-Closest}(S)$ is $O(n \log n)$.

1.4 Removing the Assumption

For this there nothing much to do. For finding the median $\overline{x}$ if we have more than one points with same $x-$coordinate which appears as the $\lfloor \frac{n}{2} \rfloor$ highest $x-$coordinate we sort only those points with respect to their $y-$coordinate update the S^x like that and then take $\lfloor \frac{n}{2} \rfloor$ highest point in S^x. We do the same for S^y and update accordingly. All this we do so that S^L and S^R has the size $\frac{n}{2}$.

Chapter 2

Median Finding in Linear Time

Median-Find(S)
Input: Set S of n distinct integers
Question: Find the $\lfloor \frac{n}{2} \rfloor^{th}$ smallest integer in S

2.1 Naive Algorithm

The naive algorithm for this will be to sort the array in $O(n \log n)$ time then return the $\lfloor \frac{n}{2} \rfloor^{th}$ element. This will take $O(n \log n)$ time. But in the next section we will show a linear time algorithm.

2.2 Linear Time Algorithm

In this section we will show an algorithm to find the median of a given set of distinct integers in $O(n)$ time complexity. Consider the following two problems:

Rank-Find (S, k)
Input: Set S of n distinct integers and an integer $k \leq n$
Question: Find the k^{th} smallest integer in S

Approximate-Split(S)
Input: Set S of n distinct integers
Question: Given S, return an integer $z \in S$ such that z where $rank(z) \in \left[\frac{n}{4}, \frac{3n}{4}\right]$

2.2.1 Solve Rank-Find using Approximate-Split

Algorithm 6: Rank-Find(S,k)

Input: Set S of n distinct integer and $k \in [n]$
Output: k^{th} smallest integer in S
begin
  if $|S| \leq 100$ **then**
    Sort S, **return** *k^{th} smallest element in S*
  $z \longleftarrow$ Approximate-Split(S) (z is the r^{th} smallest element for some $r \in \left[\frac{n}{4}, \frac{3n}{4}\right]$)
  $S_L \longleftarrow \{x \in S \mid x \leq z\}$, $S_R \longleftarrow \{x \in S \mid x > z\}$
  if $k \leq |S_L|$ **then**
    return *Rank-Find*(S_L, k)
  return *Rank-Find*$(S_R, k - |S_L|)$

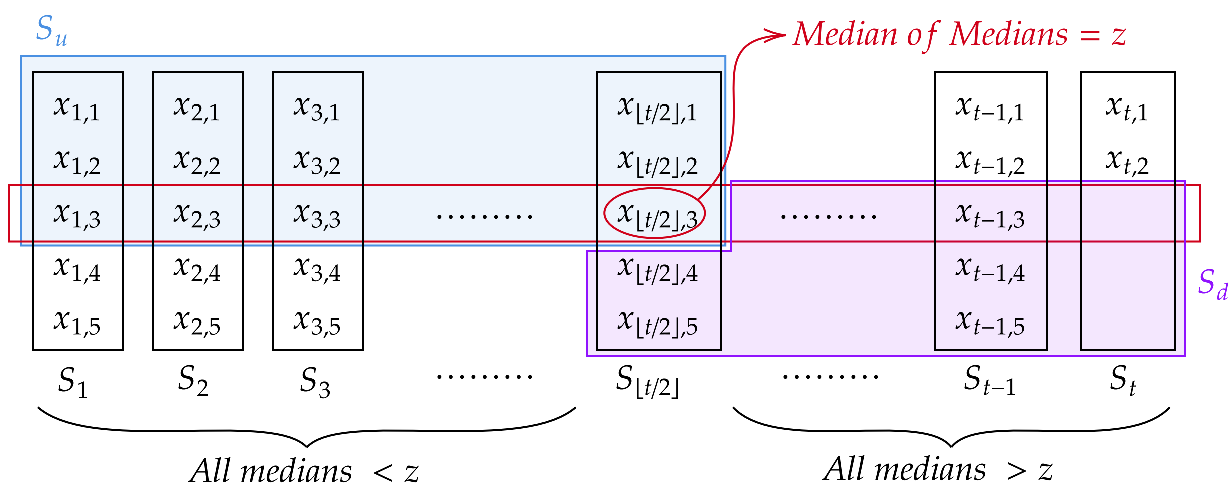
Certainly if we can solve RANK-FIND(S, k) for all $k \in [n]$ we can also solve MEDIAN-FIND. We will try to use both the problems and recurse to solve RANK-FIND in linear time.

In the above algorithm $rank(z) \in \left[\frac{n}{4}, \frac{3n}{4}\right]$. So $\frac{n}{4} \le |S_L|, |S_R| \le \frac{3n}{4}$. For now suppose RANK-FIND(S, k) takes $T_{RF}(n)$ time and APPROXIMATE-SPLIT(S) takes $T_{AS}(n)$ time. Then the time taken by the algorithm is

$$T_{RF}(n) \le O(n) + T_{AS}(n) + T_{RF}\left(\frac{3n}{4}\right)$$

2.2.2 Solve APPROXIMATE-SPLIT using RANK-FIND

We first divide S into groups of 5 elements. So take $t = \left\lceil \frac{n}{5} \right\rceil$. Now we sort each group. Since each group have constant size this can be done in $O(n)$ time. So now consider the scenario:



After sorting each of the groups we takes the medians of each group. Let z be the median of the medians. We claim that $rank(z) \in \left[\frac{n}{4}, \frac{3n}{4}\right]$.

Algorithm 7: APPROXIMATE-SPLIT(S)

Input: Set S of n distinct integers
Output: An integer $z \in S$ such that z where $rank(z) \in \left[\frac{n}{4}, \frac{3n}{4}\right]$

```
begin
    if |S| ≤ 100 then
        Sort, return Exact median
    t ⟵ ⌈n/5⌉
    S_i ⟵ i^th block of 5 elements in S for i ∈ [t − 1]
    S_t ⟵ Whatever is left in S
    for i ∈ [t] do
        Sort S_i, Let h_i be the median of S_i
    T ⟵ {h_i | i ∈ [t]}
    return RANK-FIND(T, ⌊t/2⌋)
```

So in the picture among elements in upper left the highest element is z and among the elements in lower right the lowest element is z. We will show that the number of elements smaller than z is between $\frac{n}{4}$ and $\frac{3n}{4}$. Lets call the set of elements in upper left box is S_u and the set of elements in lower right box is S_d.

Lemma 2.2.1

$|S_u|, |S_d| \geq \frac{n}{4}$

Proof: $|S_u| \geq 3 \times \left\lfloor \frac{t}{2} \right\rfloor$. For $n \geq 100$, $3 \left\lfloor \frac{t}{2} \right\rfloor > \frac{n}{4}$. Hence $|S_u| \geq \frac{n}{4}$. Now similarly $|S_d| \geq 3 \left\lfloor \frac{t}{2} - 1 \right\rfloor \geq \frac{n}{4}$. ■

Lemma 2.2.2

Number of elements in S smaller than z lies between $\frac{n}{4}$ and $\frac{3n}{4}$.

Proof: Now number of elements in S smaller than $z \geq |S_u| \geq \frac{n}{4}$. The number of elements greater than $z \geq |S_d| \geq \frac{n}{4}$. So number of elements in S smaller than $z \leq n-$ number of elements greater than $z \leq n - \frac{n}{4} = \frac{3n}{4}$. ■

Hence the APPROXIMATE-SPLIT(S) takes time

$$T_{AS}(n) = O(n) + T_{RF}\left(\frac{n}{5}\right)$$

2.2.3 Pseudocode and Time Complexity

Hence using APPROXIMATE-SPLIT the final algorithm for RANK-FIND is the following:

Algorithm 8: RANK-FIND(S,k)

Input: Set S of n distinct integer and $k \in [n]$
Output: k^{th} smallest integer in S
begin
  if $|S| \leq 100$ **then**
    Sort S, **return** k^{th} *smallest element in* S
  $t \longleftarrow \left\lceil \frac{n}{5} \right\rceil$
  $S_i \longleftarrow i^{th}$ block of 5 elements in S for $i \in [t-1]$
  $S_t \longleftarrow$ Whatever is left in S
  for $i \in [t]$ **do**
    Sort S_i, Let h_i be the median of S_i
  $T \longleftarrow \{h_i \mid i \in [t]\}$
  $z \longleftarrow$ RANK-FIND $\left(T, \left\lfloor \frac{t}{2} \right\rfloor\right)$
  $S_L \longleftarrow \{x \in S \mid x \leq z\}$, $S_R \longleftarrow \{x \in S \mid x > z\}$
  if $k \leq |S_L|$ **then**
    return *RANK-FIND*(S_L, k)
  return *RANK-FIND*$(S_R, k - |S_L|)$

Replacing $T_{AS}(n)$ in the time complexity equation of $T_{RF}(n)$ we get the equation:

$$T_{RF}(n) \leq O(n) + T_{RF}\left(\frac{n}{5}\right) + T_{RF}\left(\frac{3n}{4}\right)$$

Let $T_{RF}(n) \leq kn + +T_{RF}\left(\frac{n}{5}\right) + T_{RF}\left(\frac{3n}{4}\right)$. We claim that $T_{RF}(n) \leq cn$ for some $c \in \mathbb{N}$ for all $n \geq n_0$ where $n_0 \in \mathbb{N}$. By induction we have

$$T_{RF}(n) \leq kn + \frac{cn}{5} + \frac{3cn}{4} = \left(k + \frac{19c}{20}\right) n$$

To have $k + \frac{19c}{20} \leq c$ we have to have $k + \frac{19c}{20} \leq c \iff c \geq 20k$. So take $c \geq 20k$ and our claim follows. Hence $T_{RF}(n) = O(n)$. Since we can find any k^{th} smallest number in a given set of distinct integers in linear time we can also find the median in linear time.

Chapter 3

Polynomial Multiplication

Polynomial Multiplication

Input: Given 2 univariate polynomials of degree $n-1$ by 2 arrays of their coefficients $(a_0, \dots, a_{n-1})$ and $(b_0, \dots, b_{n-1})$ such that $A(x) = a_0 + a_1 x + \cdots + a_{n-1}x^{n-1}$ and $B(x) = b_0 + b_1 x + \cdots + b_{n-1}x^{n-1}$ respectively

Question: Given 2 polynomials of degree $n-1$ find their product polynomial $C(x) = A(x)B(x)$ of degree $2n-2$ by returning the array of their coefficients.

3.1 Naive Algorithm

We can do this naively by calculating each coefficient of C in $O(n)$ time since for any $i \in \{0, \dots, 2n-2\}$

$$c_i = \sum_{j=0}^{i} a_j b_{i-j}$$

Since there are $2n-1 = O(n)$ total coefficient of C it takes total $O(n^2)$ time. In the following section we will do this in $O(n \log n)$ time.

3.2 Strassen-Schönhage Algorithm

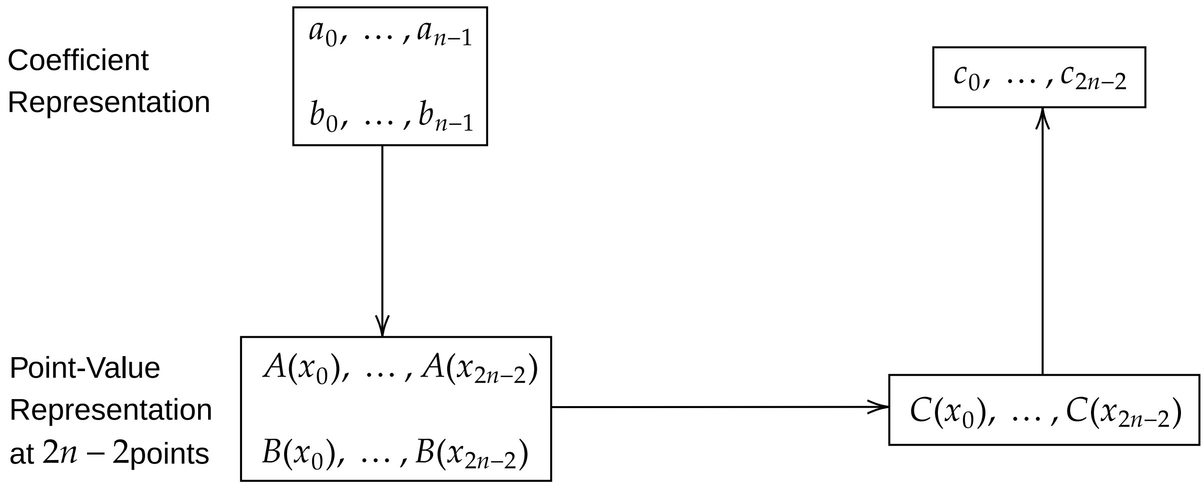
Before diving into the algorithm first let's consider how many ways we can represent a polynomial. Often changing the representation helps solving the problem in less time.

- **Coefficients**: We can represent a polynomial by giving the array of all its coefficient.
- **Point-Value Pairs**: We can evaluate the polynomial in distinct n points and give all the point-value pairs. This also uniquely represents a polynomial since there is exactly one polynomial of degree $n-1$ which passes through all these points.

Theorem 3.2.1

Given n distinct points $(x_0, y_0), \dots, (x_{n-1}, y_{n-1})$ in $\mathbb{R}^2$ there is an unique $(n-1)$–degree polynomial $P(x)$ such that $P(x_i) = y_i$ for all $i \in [\![n-1]\!]$

Since we want to find the polynomial $C(x) = A(x)B(x)$ and $C(x)$ has degree $2n-2$, we will evaluate the polynomials $A(x)$ and $B(x)$ in $2n-1$ distinct points. So we will have the algorithm like this:



3.2.1 Finding Evaluations of Multiplied Polynomial

Suppose we were given $A(x)$ and $B(x)$ evaluated at $2n-1$ distinct points $x_0, \dots, x_{2n-2}$. Then we can get $C(x)$ evaluated at $x_0, \dots, x_{2n-2}$ by

$$C(x_i) = A(x_i)B(x_i) \;\forall\; i \in [\![2n-2]\!]$$

Since there are $O(n)$ many points and for each point it takes constant time to multiply we can find evaluations of C at $x_0, \dots, x_{2n-2}$ in $O(n)$ time.

3.2.2 Evaluation of a Polynomial at Points

Question 3.1

Suppose there is only one point, x_0. Can we evaluate a $n-1$ degree polynomial $A(x) = \sum\limits_{i=0}^{n-1} a_i x^i$ at x_0 efficiently?

We can rewrite $A(x)$ as

$$A(x) = a_0 + x(a_1 + x(a_2 + x(a_3 + \cdots (a_{n-1} + x(a_n))\cdots)))$$

In this represent it is clear that we have to do n additions and n multiplications to find $A(x_0)$. Hence we can evaluate a $n-1$ degree polynomial at a point in $O(n)$ time

But we have $O(n)$ points. And if each point takes $O(n)$ time to find the evaluation of the polynomial then again it will take total $O(n^2)$ time. We are back to square one. So instead we will evaluate the polynomial in some special points and we will evaluate in all of them in $O(n\log n)$ time. So now the problem we will discuss now is to find some special n points where we can evaluate a $n-1$-degree polynomial in $O(n\log n)$ time.

Idea: Evaluate at roots of unity and use Fast Fourier Transform

Assume n is a power of 2. NWe have the polynomial $A(x) = \sum\limits_{i=0}^{n-1} a_i x^i$. So now consider the following two polynomials

$$A^0(x) = a_0 + a_2 x + a_4 x^2 + \cdots + a_{n-2} x^{\frac{n}{2}-1} \qquad A^1(x) = a_1 + a_3 x + a_5 x^2 + \cdots + a_{n-1} x^{\frac{n}{2}-1}$$

Certainly we have

$$A(x) = A^0(x^2) + xA^1(x^2)$$

Hence we can get $A(1)$ and $A(-1)$ by

$$A(1) = A^0(1) + A^1(1) \qquad A(-1) = A^0(1) - A^1(1)$$

Hence like this by evaluating two $\frac{n}{2}-1$ degree polynomials at one point we get evaluation of A at two points. More generally for any $y \geq 0$ we have

$$A(\sqrt{y}) = A^0(y) + \sqrt{y}A^1(y) \qquad A(-\sqrt{y}) = A^0(y) - \sqrt{y}A^1(y)$$

So by recursing like this evaluating at $1, -1$ we can get evaluations of A at n^{th} roots of unity.

Let

$$\omega_n^k = n^{th} \text{ root of unity for } k \in [\![n-1]\!] = e^{i\frac{k}{n}2\pi} = \cos\left(\frac{k}{n}2\pi\right) + i \sin s\left(\frac{k}{n}2\pi\right)$$

Hence we have

$$A\left(\omega_n^k\right) = A^0\left(\omega_n^{2k}\right) + \omega_n^k A^1\left(\omega_n^{2k}\right) = A^0\left(\omega_{\frac{n}{2}}^k\right) + \omega_n^k A^1\left(\omega_{\frac{n}{2}}^k\right)$$
$$A\left(-\omega_n^k\right) = A\left(\omega_n^{\frac{n}{2}+k}\right) = A^0\left(\omega_n^{2k}\right) - \omega_n^k A^1\left(\omega_n^{2k}\right) = A^0\left(\omega_{\frac{n}{2}}^k\right) - \omega_n^k A^1\left(\omega_{\frac{n}{2}}^k\right)$$

. Hence now we will solve the following problem:

RECURSIVE-DFT

Input: $(a_0, \dots, a_{n-1})$ representing $(n-1)-$degree polynomial $A(x) = \sum\limits_{i=0}^{n-1} a_i x^i$

Question: Find the evaluations of the polynomial $A(x)$ in all n^{th} roots of unity

Since A^0 and A^1 have degree $\frac{n}{2}-1$ we can use recursion. Hence the algorithm is

Algorithm 9: RECURSIVE-DFT(A)

Input: $A = (a_0, \dots, a_{n-1})$ such that $A(x) = a_0 + a_1 x + \cdots + a_{n-1}x^{n-1}$
Output: $A(x)$ evaluated at n^{th} roots of unity ω_n^k for all $k \in [\![n-1]\!]$

begin
  if $n == 1$ **then**
    return $A[0]$
  $A^0 \longleftarrow (A[0], A[2], \dots, A[n-2])$
  $A^1 \longleftarrow (A[1], A[3], \dots, A[n-1])$
  $Y^0 \longleftarrow$ RECURSIVE-DFT(A^0)
  $Y^1 \longleftarrow$ RECURSIVE-DFT(A^1)
  for $k = 0$ *to* $\frac{n}{2}-1$ **do**
    $Y[k] \longleftarrow Y^0[k] + \omega_n^k Y^1[k]$ // $A\left(\omega_n^k\right) = A^0\left(\omega_{\frac{n}{2}}^k\right) + \omega_n^k A^1\left(\omega_{\frac{n}{2}}^k\right)$
    $Y\left[k + \frac{n}{2}\right] \longleftarrow Y^0[k] - \omega_n^{\frac{n}{2}+k} Y^1[k]$ // $A\left(-\omega_n^k\right) = A^0\left(\omega_{\frac{n}{2}}^k\right) - \omega_n^k A^1\left(\omega_{\frac{n}{2}}^k\right)$
  return Y

Algorithm Time Complexity: $T(n) = 2T\left(\frac{n}{2}\right) + O(n) = O(n \log n)$.

Therefore we can evaluate a $n-1$ degree polynomial in all the n^{th} roots of unity in $O(n \log n)$ time. Hence with this algorithm we will get evaluations of the polynomial $C(x) = A(x)B(x)$ in all the $2n^{th}$ roots of unity. Now we need to interpolate the polynomial $C(x)$ from its evaluations. We will describe the process in the next subsection.

3.2.3 Interpolation from Evaluations at Roots of Unity

In this section we will show how to interpolate a $n-1$ degree polynomial from evaluations at all n^{th} roots of unity. Previously we had

$$\underbrace{\begin{bmatrix} C\left(\omega_n^0\right) \\ C\left(\omega_n^1\right) \\ C\left(\omega_n^2\right) \\ \vdots \\ C\left(\omega_n^{n-1}\right) \end{bmatrix}}_{Y} = \underbrace{\begin{bmatrix} 1 & \omega_n^0 & \omega^{0\cdot 2} & \cdots & \omega^{0\cdot(n-1)} \\ 1 & \omega_n^1 & \omega^{1\cdot 2} & \cdots & \omega^{1\cdot(n-1)} \\ 1 & \omega_n^2 & \omega^{2\cdot 2} & \cdots & \omega^{2\cdot(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_n^{n-1} & \omega^{(n-1)\cdot 2} & \cdots & \omega^{(n-1)\cdot(n-1)} \end{bmatrix}}_{V = \text{Vandermonde Matrix}} \underbrace{\begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ \vdots \\ c_{n-1} \end{bmatrix}}_{C}$$

Now vandermonde matrix is invertible since all the n^{th} roots are distinct. Therefore $C = V^{-1}Y$. But we can not do a matrix inversion to interpolate the polynomial because that will take $O(n^2)$ time. Instead we have this beautiful result:

Lemma 3.2.2
$\left(V^{-1}\right)_{j,k} = \frac{1}{n}\omega_n^{-jk}$ for all $0 \le j, k \le n-1$

Proof: Consider the matrix $n \times n$ matrix T such that $(T)_{j,k} = \frac{1}{n}\omega_n^{-jk}$. Now we will show $VT = I$ This will confirm that $V^{-1} = T$. Now

$$\sum_{k=0}^{n-1} (V)_{i,j}\,(T)_{j,k} = \sum_{k=0}^{n-1} \omega_n^{ij} \times \frac{1}{n}\omega_n^{-jk} = \frac{1}{n}\sum_{k=0}^{n-1}\left(\omega_n^{i-k}\right)^j = \begin{cases} \dfrac{1}{n}\displaystyle\sum_{k=0}^{n-1} 1 = 1 & \text{when } i = k \\ \dfrac{1}{n}\dfrac{1-\omega_n^n}{1-\omega} = 0 & \text{when } i \neq k \end{cases}$$

Hence in VT there are $1'$s on the diagonal and rest of the locations are 0. Hence $VT = I$. So $V^{-1} = T$. ■

Hence we can see the inverse of the vandermonde matrix is also a vandermode matrix with a scaling factor. We will denote $y_i = C\left(\omega_n^i\right)$ for $i \in [\![n-1]\!]$ since these values are given to us some how and we have to find the corresponding polynomial. Therefore we have

$$\underbrace{\begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ \vdots \\ c_{n-1} \end{bmatrix}}_{C} = \frac{1}{n}\underbrace{\begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega_n^{-1} & \omega^{-1\cdot 2} & \cdots & \omega^{-1\cdot(n-1)} \\ 1 & \omega_n^{-2} & \omega^{-2\cdot 2} & \cdots & \omega^{-2\cdot(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_n^{-(n-1)} & \omega^{-(n-1)\cdot 2} & \cdots & \omega^{-(n-1)\cdot(n-1)} \end{bmatrix}}_{V^{-1}} \underbrace{\begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ \vdots \\ c_{n-1} \end{bmatrix}}_{C} \underbrace{\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_{n-1} \end{bmatrix}}_{Y}$$

Observation. $nc_j = y_0 + y_1\omega_n^{-j} + y_2\omega_n^{-2j} + \cdots + y_{n-1}\omega_n^{-(n-1)j}$ *for all* $j \in [\![n-1]\!]$.

We can also see this situation as we have the polynomial $Y(x) = y_0 + y_1 x + y_2 x^2 + \cdots + y_{n-1}x^{n-1}$ and c_j is just $Y(x)$ evaluated as $\omega_n^{-j} = \omega_n^{n-j}$ multiplied by n. Hence we just reindex the n^{th} roots of unity and evaluate Y n^{th} roots of unity in $O(n \log n)$ time using the algorithm described in subsection 3.2.2

Chapter 4

Longest Increasing Subsequence

Longest Increasing Subsequence
Input: Sequence of distinct integers $A = (a_1, \ldots, a_n)$
Question: Given an array of distinct integers find the longest increasing subsequence i.e. return maximum size set $S \subseteq [n]$ such that $\forall\, i, j \in S, i < j \implies a_i < a_j$

Definition 4.1: Dynamic Programming

Dynamic Programming has 3 components:

1. Optimal Substructure: Reduce problem to smaller independent problems
2. Recursion: Use recursion to solve the problems by solving smaller independent problems
3. Table Filling: Use a table to store the result to solved smaller independent problems.

4.1 $O(n^2)$ Time Algorithm

Given $A = (a_1, \ldots, a_n)$ first we will create a n-length array where i^{th} entry stores the length and longest increasing subsequence ending at a_i. Certainly we have the following recursion relation

$$\text{LIS}(k) = 1 + \max_{j<k,\ a_j<a_k} \{\text{LIS}(j)\}$$

since if a subsequence $S \subsetneq [n]$ is the longest increasing subsequence ending at a_k then certainly $S - \{k\}$ is the longest increasing subsequence which ends at $a_j < a_k$ for some $j < k$.

Hence in the table we start with 1st position and using the recursion relation we fill the table from left. And after the table is filled we look for which entry of the table has maximum length. So the algorithm will be following:

Algorithm 10: LIS(A)

Input: Sequence of distinct integers $A = (a_1, \ldots, a_n)$
Output: Maximum size set $S \subseteq [n]$ such that $\forall\, i, j \in S, i < j \implies a_i < a_j$.

```
begin
    Create an array T of length n
    for i ∈ [n] do
        T[i][1] ⟵ 1 + max{T[j][1] : j < k, a_j < a_k}        // Finds LIS[i]
        T[i][2] ⟵ T[T[i][1] − 1][2]
    Index ⟵ max{T[j][1] : j ∈ [n]}
    return T[Index]
```

For each iteration of the loop it takes $O(n)$ time to find LIS$[i]$. Hence the time complexity of this algorithm is $O(n^2)$.

4.2 $O(n \log n)$ Time Algorithm

In the following algorithm we update the longest increasing sequence every time we see a new element of the given sequence. At any time we keep the best available sequence.

Algorithm 11: QuickLIS(A)

Input: Sequence of distinct integers $A = (a_1, \ldots, a_n)$
Output: Maximum size set $S \subseteq [n]$ such that $\forall\, i, j \in S, i < j \implies a_i < a_j$.

begin
- Create an array T of length n with all entries 0
- Create an array M of length n
- **for** $i = 1, \ldots, n$ **do**
  - $M[i] \longleftarrow \infty$
- **for** $i = 1, \ldots, n$ **do**
  - $k \longleftarrow$Find smallest index i such that $M[k] > a_i$ using Binary-Search
  - $M[k] \longleftarrow i$
  - $T[i] \longleftarrow M[k-1]$ `// Pointer to the previous element of the sequence`
- $k_0 \longleftarrow$ Largest k_0 such that $M[k_0]$ is finite
- Create an array S of length k_0
- **for** $i = k_0, \ldots, 1$ **do**
  - **if** $i = k_0$ **then**
    - $S[k_0] \longleftarrow M[k_0]$
    - Continue
  - $S[i] \longleftarrow T\big[S[i+1]\big]$ `//` $T[S[i+1]]$ `is pointer to previous value of sequence`
- **return** (k_0, S)

Lemma 4.2.1
For any index $M[k]$ is non increasing

Proof: Every time we change a value of $M[k]$ we replace by something smaller. So $M[k]$ is non increasing. ■

Lemma 4.2.2
At any time t, $M[1] \le M[2] \le \cdots \le M[t]$

Proof: content... ■

Chapter 5

Opimal Binary Search Tree

Chapter 6

Hoffman Encoding