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

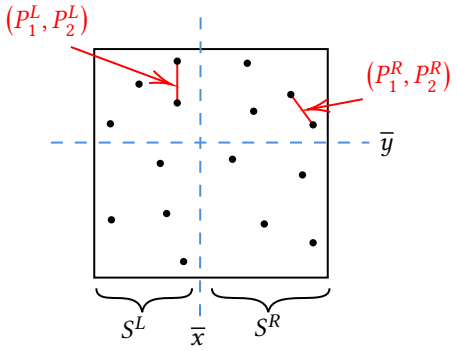
- Divide: Divide the problem into two parts (roughly equal)
- Conquer: Solve each part individually recursively. If the subproblem sizes are small enough, however, just solve the subproblems in a straightforward manner.
- Combine: Combine the solutions to the subproblems into the solution.

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.

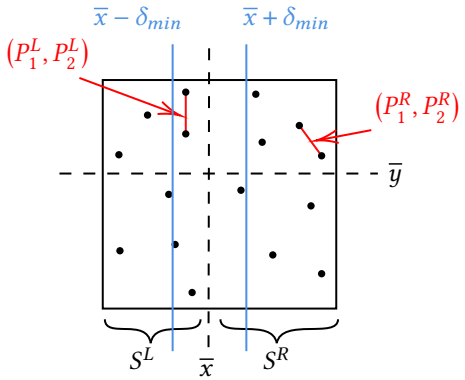
Question 1.1: We are not done

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 $\in [\bar{x} - \delta_{min}, \bar{x} + \delta_{min}]$.
- $|y_i - y_j| \leq \delta_{min}$

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



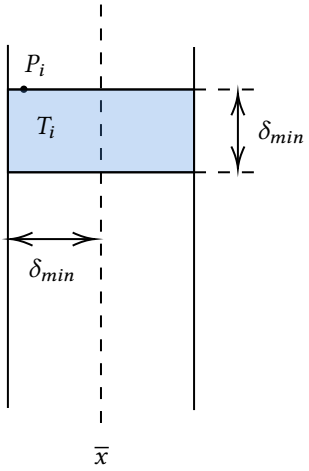
We now sort all the points in the T by their decreasing $y-$coordinate. Let T_y be the array of points. For each $P_i \in T_y$ define the region

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

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| ≤ 10 then
3          Solve by Brute Force (Consider every pair of points)
4      S^x ⟵ S sorted by x−coordinate,        S^y ⟵ S sorted by y−coordinate
5      x̄ ⟵ ⌊n/2⌋ highest x−coordinate,        ȳ ⟵ ⌊n/2⌋ highest y−coordinate
6      S^L ⟵ {P_i | x_i < x̄, ∀ i ∈ [n]},      S^R ⟵ {P_i | x_i ≥ x̄, ∀ i ∈ [n]}
7      (P_1^L, P_2^L, δ^L) ⟵ FIND-CLOSEST(S^L),      (P_1^R, P_2^R, δ^R) ⟵ FIND-CLOSEST(S^R)
8      δ_min ⟵ min{δ^L, δ^R}
9      if δ_min < δ^L then
10         P_1 ⟵ P_1^R, P_2 ⟵ P_2^R
11     else
12         P_1 ⟵ P_1^L, P_2 ⟵ P_2^L
13     T ⟵ {P_i | |x_i − x̄| ≤ δ_min}
14     T_y ⟵ T sorted by decreasing y−coordinate
15     for P ∈ T_y do
16         U ⟵ Next 8 points
17         for P̂ ∈ U do
18             if d(P, P̂) < δ_min then
19                 δ_min ⟵ d(P, P̂)
20                 (P_1, P_2) ⟵ (P, P̂)
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: 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 $\delta = d(P_i, P_j)$

1 **begin**
2 **if** $|S| \leq 10$ **then**
3 Solve by Brute Force
4 $\overline{x} \longleftarrow \lfloor \frac{n}{2} \rfloor$ highest $x-$coordinate
5 $\overline{y} \longleftarrow \lfloor \frac{n}{2} \rfloor$ highest $y-$coordinate
6 $S^L \longleftarrow \{P_i \in S^x \mid x_i < \bar{x},\ \forall\ i \in [n]\}$
7 $S^L_y \longleftarrow \{P_i \in S^y \mid x_i < \overline{x}\}$
8 $S^R \longleftarrow \{P_i \in S^x \mid x_i \geq \bar{x},\ \forall\ i \in [n]\}$
9 $S^R_y \longleftarrow \{P_i \in S^y \mid x_i \geq \overline{x}\}$
10 $(P^L_1, P^L_2, \delta^L) \longleftarrow$ Find-Closest-Sorted(S^L, S^L_y)
11 $(P^R_1, P^R_2, \delta^R) \longleftarrow$ Find-Closest-Sorted(S^R, S^R_y)
12 $\delta_{min} \longleftarrow \min\{\delta^L, \delta^R\}$
13 **if** $\delta_{min} < \delta^L$ **then**
14 $P_1 \longleftarrow P^R_1, P_2 \longleftarrow P^R_2$
15 **else**
16 $P_1 \longleftarrow P^L_1, P_2 \longleftarrow P^L_2$
17 $T \longleftarrow \{P_i \mid |x_i - \overline{x}| \leq \delta_{min}\}$
18 $T_y \longleftarrow$ Reverse$(\{P_i \in S^y \mid |x_i - \overline{x}| \leq \delta_{min}\})$
19 **for** $P \in T_y$ **do**
20 $U \longleftarrow$ Next 8 points
21 **for** $\hat{P} \in U$ **do**
22 **if** $d(P, \hat{P}) < \delta_{min}$ **then**
23 $\delta_{min} \longleftarrow d(P, \hat{P})$
24 $(P_1, P_2) \longleftarrow (P, \hat{P})$
25 **return** (P_1, P_2, δ_{min})

Algorithm 5: 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
4 $S^x \longleftarrow S$ sorted by $x-$coordinate
5 $S^y \longleftarrow S$ sorted by $y-$coordinate
6 **return** Find-Closest-Sorted(S^x, S^y)

This algorithm only sorts one time. So time complexity for 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 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

```
1 begin
2     if |S| ≤ 100 then
3         Sort S, return k^th smallest element in S
4     z ⟵ APPROXIMATE-SPLIT(S)        (z is the r^th smallest element for some r ∈ [n/4, 3n/4])
5     S_L ⟵ {x ∈ S | x ≤ z}, S_R ⟵ {x ∈ S | x > z}
6     if k ≤ |S_L| then
7         return RANK-FIND(S_L, k)
8     return RANK-FIND(S_R, k − |S_L|)
```

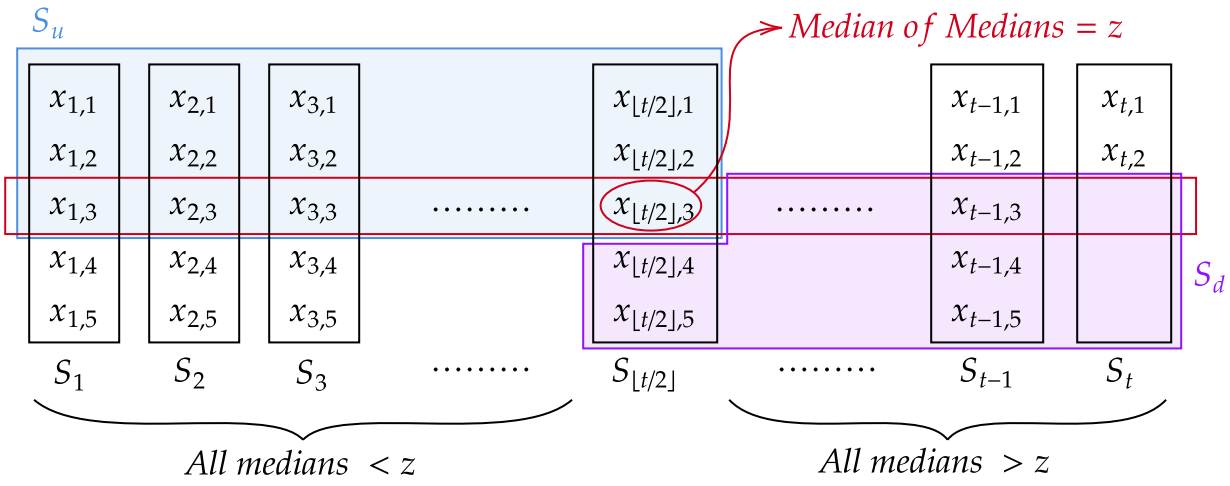
Certainly if we can solve RANK-FIND(S, k) for all $k \in [n]$ we can also solve MEDIAN-FIND. We will try to use both the problems and recurse to solve RANK-FIND in linear time.

In the above algorithm $rank(z) \in \left[\frac{n}{4}, \frac{3n}{4}\right]$. So $\frac{n}{4} \leq |S_L|, |S_R| \leq \frac{3n}{4}$. 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

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

2.2.2 Solve APPROXIMATE-SPLIT using RANK-FIND

We first divide S into groups of 5 elements. So take $t = \lceil \frac{n}{5} \rceil$. 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 $rank(z) \in \left[\frac{n}{4}, \frac{3n}{4}\right]$

```
1  begin
2      if |S| ≤ 100 then
3          Sort, return Exact median
4      t ⟵ ⌈n/5⌉
5      S_i ⟵ i^th block of 5 elements in S for i ∈ [t − 1]
6      S_t ⟵ Whatever is left in S
7      for i ∈ [t] do
8          Sort S_i, Let h_i be the median of S_i
9      T ⟵ {h_i | i ∈ [t]}
10     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

```
1  begin
2      if |S| ≤ 100 then
3          Sort S, return k^th smallest element in S
4      t ⟵ ⌈n/5⌉
5      S_i ⟵ i^th block of 5 elements in S for i ∈ [t − 1]
6      S_t ⟵ Whatever is left in S
7      for i ∈ [t] do
8          Sort S_i, Let h_i be the median of S_i
9      T ⟵ {h_i | i ∈ [t]}
10     z ⟵ Rank-Find(T, ⌊t/2⌋)
11     S_L ⟵ {x ∈ S | x ≤ z}, S_R ⟵ {x ∈ S | x > z}
12     if k ≤ |S_L| then
13         return Rank-Find(S_L, k)
14     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 $(a_0, \ldots, a_{n-1})$ and $(b_0, \ldots, b_{n-1})$ 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 $i \in \{0, \ldots, 2n-2\}$

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

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

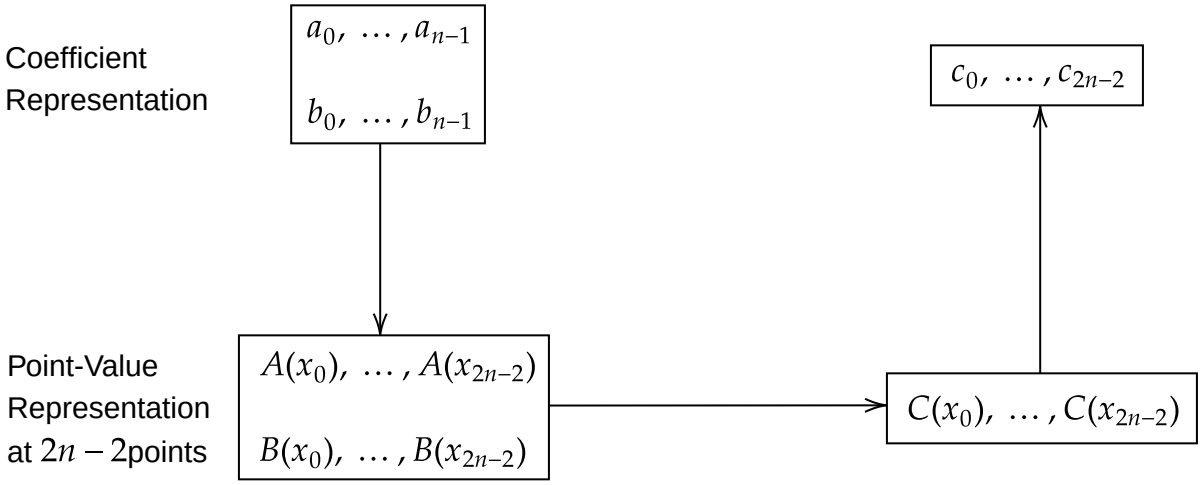
Given n distinct points $(x_0, y_0), \ldots, (x_{n-1}, y_{n-1})$ in $\mathbb{R}^2$ there is an unique $(n-1)-$degree polynomial $P(x)$ such that $P(x_i) = y_i$ for all $i \in [\![n-1]\!]$

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 $x_0, \ldots, x_{2n-2}$. Then we can get $C(x)$ evaluated at $x_0, \ldots, x_{2n-2}$ 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 $x_0, \ldots, x_{2n-2}$ 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 $A(x) = \sum_{i=0}^{n-1} a_i x^i$ 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 $A(x) = \sum_{i=0}^{n-1} a_i x^i$. 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: $(a_0, \ldots, a_{n-1})$ representing $(n-1)-$degree polynomial $A(x) = \sum_{i=0}^{n-1} a_i x^i$

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: $A = (a_0, \ldots, a_{n-1})$ 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 $k \in [\![n-1]\!]$

```
1  begin
2      if n == 1 then
3          return A[0]
4      A^0 ⟵ (A[0], A[2], ..., A[n-2])
5      A^1 ⟵ (A[1], A[3], ..., A[n-1])
6      Y^0 ⟵ RECURSIVE-DFT(A^0)
7      Y^1 ⟵ RECURSIVE-DFT(A^1)
8      for k = 0 to n/2 - 1 do
9          Y[k] ⟵ Y^0[k] + ω_n^k Y^1[k]                    // A(ω_n^k) = A^0(ω_{n/2}^k) + ω_n^k A^1(ω_{n/2}^k)
10         Y[k + n/2] ⟵ Y^0[k] - ω_n^{n/2+k} Y^1[k]        // A(-ω_n^k) = A^0(ω_{n/2}^k) - ω_n^k A^1(ω_{n/2}^k)
11     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

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

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

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

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 $Y(x) = y_0 + y_1x + y_2x^2 + \cdots + y_{n-1}x^{n-1}$ 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 4.2.2

Chapter 4

Dynamic Programming

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 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 $\forall\, i, j \in S, i < j \implies a_i < a_j$

4.1.1 $O(n^2)$ 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 $S \subseteq [n]$ 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 $\forall\, i, j \in S, i < j \implies a_i < a_j$.

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

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

4.1.2 $O(n \log n)$ 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.

Idea. *We can make an increasing subsequence longer by picking the smallest number for position k so that there is an increasing subsequence of length k. Doing this we can maximize the length of the subsequence.*

Theorem 4.1.1

Is $S \subseteq A$ is the longest increasing subsequence of length t then for any $k \in [t]$ the number $S(k)$ is the smallest number in subarray of A starting at first and ending at $S(k)$ such that there is an increasing subsequence of length k ending at $S(k)$.

Proof: Suppose $\exists\ k \in [t]$ such that k is the smallest number in $[t]$ such that $S(k)$ is not the smallest number to satisfy the condition. Now denote the subarray of A starting at first and ending at $S(k)$ by A_k. Now let $x \in A_k$ be the smallest number in A_k such that there is an increasing subsequence of length k ending at x. Certainly $x < S(k)$. Now since k is the smallest index which does not satisfy the given condition, $\forall\ j \in [k-1]$, $S(j)$ is the smallest number in A_j such that there is an increasing subsequence of length j ending at $S(j)$. Then consider the subsequence $\{S(1), \ldots, S(k-1), x, S(k), S(k+1), \ldots, S(t)\}$. This is an increasing subsequence of A and has length $t+1$. But this contradicts the minimality of S. Hence contraditon. Every element of S follows the given condition. ■

So we will construct an increasing subsequence by gradually where each step this property is followed, i.e. at each step we will ensure that the sequence built at some time have the above propert. So now we describe the algorithm.

Algorithm 11: QuickLIS(A)

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

```
1  begin
2      Create an array T of length n with all entries 0
3      Create an array M of length n
4      for i = 1, ..., n do
5          M[i] ⟵ ∞
6      for i = 1, ..., n do
7          k ⟵Find smallest index i such that M[k] > a_i using Binary-Search
8          M[k] ⟵ i
9          T[i] ⟵ M[k − 1]        // Pointer to the previous element of the sequence
10     l ⟵ Largest l such that M[l] is finite
11     Create an array S of length l
12     for i = l, ..., 1 do
13         if i = l then
14             S[l] ⟵ M[l]
15             Continue
16         S[i] ⟵ T[S[i + 1]]     // T[S[i+1]] is pointer to previous value of sequence
17     return (l, S)
```

Time Complexity: To create the arrays and the first for loop takes $O(n)$ time. In each iteration of the for loop at line 6 it takes $O(\log n)$ time to find k and rest of the operations in the loop takes constant time. So the for loop takes $O(n \log n)$ time. Then To find l and creating S it takes $O(n)$ time. Then in the for loop at line 12 in each iteration it takes constant time. So the for loop at line 12 takes in total $O(n)$ time. Therefore the algorithm takes $O(n \log n)$ time.

We will do the proof of correctness of the algorithm now.

Lemma 4.1.2

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. ■

We denote the state of array M at i^{th} iteration by M^i. Then we have the following lemma:

Lemma 4.1.3

At any time i, $M^i[1] \leq M^i[2] \leq \cdots \leq M^i[n]$

Proof: We will prove this by induction on i. The base case follows naturally. Now for i^{th} iteration suppose $M^i[k]$ is replaced by x_i. Then we know $\forall\ j < k$ we have $M^i[j] \leq x_i$. By inductive hypothesis at time $t-1$ we have M as an increasing sequence. Now before replacing $M^i[k] \leq M^i[k+1] \leq \cdots M^i[n]$. Now by Lemma 5.1.2 $M^i[k]$ is nonincreasing. So So we still have $M^i[1] \leq \cdots M^i[k-1] \leq x_i \leq M^i[k+1] \leq \cdots \leq M^i[n]$. Hence bt mathematical induction it holds. ■

Now suppose at i^{th} iteration k_i is largest such that $M^i[k_i] < \infty$. Then S^i denote the set constructed like the way we constructed at line 12–16 in the algorithm i.e.

$$S^i[k_i] = M^i[k_i] \qquad \text{and} \qquad S^i[j] = T[S^i[j+1]] \quad \forall\ j \in [k_i - 1]$$

Lemma 4.1.4

After any i^{th} iteration, for $k \in [n]$ if $M^i[k] < \infty$ then $S^i[k]$ stores the smallest value in $x_1, \ldots, x_i$ such that there is an increasing subsequence of size k that ends in $S^i[k]$.

Proof: We will induction on i. Base case: This is true after first iteration since only $M^1[1] < \infty$. So this naturally follows.

Suppose this is true after i iterations. Now at $(i+1)^{th}$ iteration suppose t be the smallest index such that $M^i[t] > x_{i+1}$. Then we have

$$M^i[1], \ldots, M^i[t-1] < x_{i+1} < M^i[k], \ldots, M^i[n] \implies S^i[1], \ldots, S^i[t-1] < x_{i+1} < S^i[k], \ldots, S^i[k_i]$$

Now for $k \leq t-1$ it is true by the inductive hypothesis. For $k > t$ and if $M^{i+1}[k] < \infty$ then $S^{i+1}[k]$ is the smallest value in $x_1, \ldots, x_{i+1}$ such that there is an increasing subsequence of size k that ends in $S^{i+1}[k]$ since this was true for i^{th} iteration.

Now only the case when $k = t$ is remaining. If $S^{i+1}[k]$ is not the smallest value in $x_1, \ldots, x_{i+1}$ to have an increasing subsequence of size k ending at $S^{i+1}[k]$ then let x_j was the smallest value to satisfy this condition where $j < i+1$. Then naturally $x_j < x_{i+1}$. Then $M^i[t] \leq x_j < x_{i+1}$. But we t was the smallest number such that $M^i[t] > x_{i+1}$. Hence contradiction. Therefore $S^i[k]$ is the smallest value in $x_1, \ldots, x_{i+1}$ to have an increasing subsequence of size k ending at $S^{i+1}[k]$.

Therefore by mathematical induction this is true for all iterations. ■

Theorem 4.1.5

S is the longest increasing subsequence of A.

Proof: After the n^{th} iteration $S^n = S$ and $k_n = l$. Hence by Lemma 5.1.4 we can say for all $k \in [l]$, $S[k]$ is the smallest number such that there is an increasing sequence of length k ending at $S[k]$. Now we want to show that this increasing sequence is the longest increasing subsequence of A. Suppose S is not the longest increasing subsequence. Let T be the longest increasing subsequence of length t. Then suppose $j \leq l$ be the smallest index such that $S[j] \neq T[j]$. Now $S[j]$ is the smallest number in $x_1, \ldots, x_n$ such that there is an increasing subsequence of length j ending at $S[j]$. Hence we have $S[j] < T[j]$. Now for all $i < j$ we have $S[i] = T[i]$. Then we form this new subsequence $\hat{T} = \{T[1], T[2], \ldots, T[j-1], S[j], T[j], \ldots, T[t]\}$. Certainly $\hat{T}$ has length $t+1$ and it is also an increasing subsequence. But this contradicts the maximality condition of T. Hence S is indeed the longest increasing subsequence. ■

4.2 Optimal Binary Search Tree

OptimalBST

Input: A sorted array $A = (a_1, \ldots, a_n)$ of search keys and an array of their probability distributions $P = (p(a_1), \ldots, p(a_n))$

Question: Given array of keys A and their probabilities the probability of accessing a_i is $p(a_i)$ then return a binary tree with the minimum cost where for any binary tree T, $\text{Cost}(T) = \sum_{i=1}^{n} p(a_i) \cdot height_T(a_i)$.

So let T be the optimal binary search tree with a_k as its root for some $k \in [n]$. Let T_l and T_r denote the tree rooted at the left child and right child of a_k in T respectively. Then:

$$\text{Cost}(T) = p_k + \sum_{i<k} p_i \left(1 + height_{T_l}(a_i)\right) + \sum_{i>k} p_i \left(1 + height_{T_r}(a_i)\right) = \sum_{i=1}^{n} p_i + \underbrace{\sum_{i<k} p_i \cdot height_{T_l}(a_i)}_{\text{Cost}(T_l)} + \underbrace{\sum_{i>k} p_i \cdot height_{T_l}(a_i)}_{\text{Cost}(T_r)}$$

We will use the use of notion in general $\text{OPTCost}(i,k) = \text{Cost}(T_i^k)$ where T_i^k is the optimal binary tree of the subarray $A[i \ldots k]$ for any $i \le k \le n$. Therefore we arrive at the following recurrence relation

$$\text{OPTCost}(i,k) = \begin{cases} 0 & \text{when } i > k \\ \sum_{j=i}^{k} p(a_j) + \min_{i \le r \le k} \{\text{OPTCost}(i, r-1) + \text{OPTCost}(r+1, k)\} & \text{otherwise} \end{cases}$$

So the algorithm for constructing the optimal binary search tree is following:

Algorithm 12: OptimalBST(A, P)

Input: A sorted array $A = (a_1, \ldots, a_n)$ of search keys and an array of their probability distributions $P = (p(a_1), \ldots, p(a_n))$

Output: Binary Tree T with the minimum search cost, $\text{Cost}(T) = \sum_{i=1}^{n} p(a_i) \cdot height_T(a_i)$

```
1  begin
2      for i = 1, ..., n do
3          OPTCost[i, i] ⟵ (p(a_i), a_i), OPTCost[0, i] ⟵ (0, None)
4      for d = 2, ..., n do
5          for i ∈ [n + 1 − d] do
6              minval ⟵ 0
7              for k = i + 1, ..., i + d − 2 do
8                  newval ⟵ OPTCost[i, k − 1][1] + OPTCost[k + 1, i + d − 1][1]
9                  if minval > newval then
10                     minval ⟵ newval
11                     Index ⟵ k
12             OPTCost[i, i + d − 1] ⟵ (minval + Σ_{k=1}^{i+d−1} p(a_k), k)
13             a_k.left ⟵ OPTCost[i, k − 1][2]
14             a_k.right ⟵ OPTCost[k + 1, i + d − 1][2]
15     return OPTCost[1, n]
```

Time Complexity: To two for loops ar line 4 and line 5 takes $O(n^2)$ many iterations. Now the inner most for loop at line 7 runs $O(n)$ iterations where in each iterations it takes constant runtime. So the total running time of the algorithm is $O(n^3)$.

Chapter 5

Greedy Algorithm

5.1 Maximal Matching

Maximal-Matching
Input: Graph $G = (V, E)$
Question: Find a maximal matching $M \subseteq E$ of G

Before diving into the algorithm to find a matching or maximal matching we first define what is a matching.

Definition 5.1.1: Matching

For a graph $G = (V, E)$ a matching $M \subseteq E$ is a set of edges such that no two edges in M are incident on same vertex.

Definition 5.1.2: Maximal Matching

For a graph $G = (V, E)$ a matching $M \subseteq E$ is maximal if it cannot be extended and still by adding an edge.

There is also a maximum matching which can be easily understood from the name:

Definition 5.1.3: Maximum Matching

For a graph $G = (V, E)$ a matching $M \subseteq E$ is maximum if it is maximal and has the maximum size among all the maximal matchings.

Idea. *The idea is to create a maximal matching we will just go over each edge one by one and check if after adding them to the set M the matching property still holds.*

Algorithm 13: Maximal-Matching

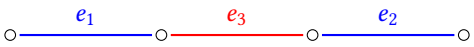
Input: Graph $G = (V, E)$
Output: Maximal Matching $M \subseteq E$ of G

```
1 begin
2     M ⟵ ∅
3     Order the edges E = {e_1, ..., e_k} arbitrarily
4     for e ∈ E do
5         if M ∪ {u} is matching then
6             M ⟵ M ∪ {e}
7     return M
```

Question 5.1

Do we always get the largest possible matching?

Solution: Clearly algorithm output is not optimal always. We get a maximal matching sure. But we don't get a maximum matching always. For example the following graph



If we start from e_1 we get the matching $\{e_1.e_2\}$ which is maximum matching but if we start from e_3 then we get only the maximal matching $\{e_3\}$ which is not maximum. ■

Since the algorithm output may not be optimal always we can ask the following question

Question 5.2

How large is the matching obtained compared to the maximum matching?

This brings us to the following result:

Theorem 5.1.1

For any graph G let the greedy algorithm obtains the matching M and the maximum matching is $M^\star$. Then

$$|M| \geq \frac{1}{2}|M^\star|$$

Proof: Consider an edge $e \in M^\star$ but $e \notin M$. Since e wasn't picked in M, $\exists\, e' \in M \setminus M^\star$ such that e and e' are incident on same vertex. Thus define the function $f : M^\star \to M$ where

$$f(e) = \begin{cases} e & \text{when } e \in M \\ e' & \text{when } e \in M^\star \setminus M \text{ where } e' \in M \setminus M^\star \text{ such that } e' \cap e \neq \emptyset \end{cases}$$

Now note that there are at most two edges in $M^\star$ that are adjacent to an edge $e' \in M$ which will be mapped to e'. Hence

$$|M \setminus M^\star| \geq \frac{1}{2}|M^\star \setminus M|$$

Therefore $|f^{-1}(e')| \leq 2 \;\forall\, e' \in M$. Hence

$$|M^\star| = |M \cap M^\star| + |M^\star \setminus M| \leq |M \cap M^\star|| + 2|M \setminus M^\star| \leq 2|M|$$

Therefore we have the result $|M| \geq \frac{1}{2}|M^\star|$. ■

Alternate Proof : Let M_1 and M_2 are two matchings. Consider the symmetric difference $M_1 \triangle M_2$. This consists of edges that are in exactly one of M_1 and M_2. Now in $M \triangle M^\star$ we have the following properties:

(a) Every vertex in $M \triangle M^\star$ has degree $\leq 2 \implies$ Each component is a path or an even cycle.

(b) The edges of M and $M^\star$ alternate.

Now we will prove the following property about the connected components of $M \triangle M^\star$.

Claim: No connected component is a single edge.
Proof: This is because let e be a connected component. So the two edges e_1, e_2 which are adjacent to e, they are either in both M and $M^\star$ or not in M and $M^\star$. The former case is not possible because then e_1, e_2, e are all in either M or $M^\star$ which is not possible as they do not satisfy the condition of matching. For the later case since $M^\star$ is maximal matching, $e \in M^\star$. Then $e \notin M$. That means $e, e_1, e_2 \notin M$ which is not possible since M is also a maximal matching. Therefore no connected component is a single edge. ■

Therefore every path has length ≥ 2. Therefore ratio of # edges of M to # edges of $M^\star$ in a path is ≤ 2. And for cycles we have # edges of M = # edges of $M^\star$. So in every connected component C of $M \triangle M^\star$ the ratio $\frac{|M^\star \cap C|}{|M \cap C|} \leq 2$. Therefore we have

$$\frac{|M^\star|}{|M|} = \frac{|M \cap M^\star| + \sum_C |M^\star \cap C|}{|M \cap M^\star| + \sum_C |M \cap C|} \leq 2$$

Hence we have $|M| \geq \frac{1}{2}|M^\star|$.

■

5.2 Huffman Encoding

HUFFMAN-CODING
Input: n symbols $A = (a-1, \ldots, a_n)$ and their frequencies $P = (f_1, \ldots, f_n)$ of using symbols
Question: Create a binary encoding such that:

- Prefix Free: The code for one word can not be prefix for another code
- Minimality: Minimize $\text{COST}(b) = \sum_{i=1}^{n} f_i \cdot \text{LEN}(b(a_i))$ where $b : A \to \{0,1\}^*$ is the binary encoding

Assignment of binary strings can also be scene as placing the symbols in a binary tree where at any node 0 means left child and 1 means right child. Then the first condition implies that there can not be two codes which lies in the same path from the root to a leaf. I.e. it means that all the codes have to be in the leaves. Then the length of the binary coding for a symbol is the height of the symbol in the binary tree.

We can think the frequencies as the probability of appearing for a letter. We denote the probability of appearing of the letter a_i by $p(a_i) \coloneqq \frac{f_i}{\sum_{i=1}^{n} f_i}$. So the we can see the updated cost function

$$\text{COST}(b) = \sum_{i=1}^{n} p(a_i) \cdot \text{LEN}(b(a_i))$$

And from now on we will see the frequencies as probabilities and cost function like this

5.2.1 Optimal Binary Encoding Tree Properties

Then our goal is to finding a binary tree with minimum cost where all the symbols are at the leaves. We have the following which establish the optimality of Huffman encoding over all prefix encodings where each symbol is assigned a unique string of bits.

Lemma 5.2.1
In the optimal encoding tree least frequent element has maximum height.

Proof: Suppose that is not the case. Let T be the optimal encoding tree and let the least frequent element x is at height h_1 and the element with the maximum height is y with height h_2 and we have $h_1 < h_2$. Then we construct a new encoding tree T' where we swap the positions of x and y. So in T' height of y is h_1 and height of x is h_2. Then

$$\text{COST}(T) - \text{COST}(T') = (p(x)h_1 + p(y)h_2) - (p(x)h_2 + p(y)h_1) = (p(x) - p(y))(h_1 - h_2)$$

Since $p(x) < p(y)$ and $h_1 < h_2$ we have $\text{COST}(T) - \text{COST}(T') > 0$. But that is not possible since T is the optimal encoding tree. So T should have the minimum cost. Hence contradiction. x has the maximum height. ■

Lemma 5.2.2

The optimal encoding binary tree must be complete binary tree. (i.e. every non-leaf node has exactly 2 children)

Proof: Suppose T be the optimal binary tree and there is a non-leaf node r which has only one child at height h. By Lemma 6.2.1 the least frequent element x has the maximum height, h_m.

Then consider the new tree $\hat{T}$ where we place the least frequent element at height h and make it the second child of the node r. Then

$$\text{Cost}(T) - \text{Cost}(\hat{T}) = p(x)h_m - p(x)h = p(x)(h_m - h) > 0$$

But this is not possible as T is the optimal binary tree and it has the minimal cost. Hence contradiction. Therefore the optimal encoding binary tree must be a complete binary tree. ■

Lemma 5.2.3

There is an optimal binary encoding tree such that the least frequent element and the second least frequent element are siblings at the maximum height.

Proof: Let T be optimal binary encoding tree. Suppose x, y are the least frequent element and the second least frequent element. And suppose b, c be two siblings at the maximum height of the tree (There may be many such siblings, and if so pick any such pair.). If $\{x, y\} = \{b, c\}$ we are done. So suppose not. Let the frequencies of x, y, b, c are respectively $p(x), p(y), p(b), p(c)$ and heights of x, y, b are h_x, h_y and h respectively. WLOG assume $p(x) \leq p(y)$ and $p(b) \leq p(c)$.

Now since we know x, y have the smallest frequencies we have $p(x) \leq p(b)$ and $p(y) \leq p(c)$. And since b, c have the maximum height we have $h_x, hy \geq h$. So we switch the position of x with b to form the new tree T'. And from T' we swap the positions fo y and c to form a new tree T''.

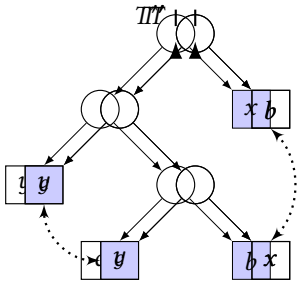


Figure 5.1: Showing that the lowest probability nodes are siblings at the tree's lowest level.

Now we will calculate how the cost changes as we go from T to T' and T' to T''. First check for $T \to T'$. Almost all the nodes contribute the same except x, b. So we have

$$\text{Cost}(T) - \text{Cost}(T') = (h_x \cdot p(x) + h \cdot p(b)) - (h_x \cdot p(b) + h \cdot p(x)) = (p(b) - p(x))(h - h_x) \geq 0$$

Therefore swapping x and b does not increase the cost and since T is the optimal binary encoding tree the cost doesn't decrease either. Therefore the costs are equal. Hence T' is also an optimal tree.

Similarly we calculate cost for going from T' to T'' we have

$$\text{Cost}(T') - \text{Cost}(T'') = (h_y \cdot p(y) + h \cdot p(c)) - (h_y \cdot p(c) + h \cdot p(y)) = (p(c) - p(y))(h - h_y) \geq 0$$

Therefore swapping y and c also does not increase the cost and since T' is the optimal binary encoding tree the cost doesn't decrease either. Therefore the costs are equal. Hence T'' is also an optimal tree. Hence T'' is the optimal tree where the least frequent element and second last frequent element are siblings. ■

By the Lemma 6.2.2 and Lemma 6.2.3 we have that the least frequent element and the second least frequent element are siblings and they have the maximum height.

Observation. *The cost of the trees T_n and T_{n-1} differ only by the fixed term $p(z) = p(x) + p(y)$ which does not depend on the tree's structure. Therefore minimizing the cost for T_n is equivalent to minimizing the cost of T_{n-1}.*

Theorem 5.2.4

Given an instance with symbols $\mathcal{I}$:

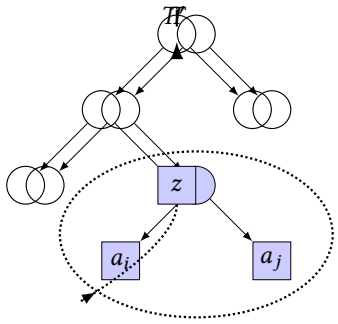
$$\begin{array}{cccccccc} a_1, & a_2, & \cdots, & a_i, & \cdots, & a_j, & \cdots, & a_n \\ p(a_1), & p(a_2), & \cdots, & p(a_i), & \cdots, & p(a_j), & \cdots, & p(a_n) \end{array} \quad \text{with probabilities}$$

such that a_i, a_j are the least frequent and second least frequent elements respectively. Consider the instance with $n-1$ symbols $\mathcal{I}'$:

$$\begin{array}{ccccccccccc} a_1, & a_2, & \cdots, & a_{i-1}, & a_{i+1}, & \cdots, & a_{j-1}, & a_{j+1}, & \cdots, & a_n, & z \\ p(a_1), & p(a_2), & \cdots, & p(a_{i-1}), & p(a_{i+1}) & \cdots, & p(a_{j-1}), & p(a_{j+1}), & \cdots, & p(a_n), & p(a_i)+p(a_j) \end{array}$$

Let T' be the optimal tree for this instance $\mathcal{I}'$. Then there is an optimal tree for the original instance $\mathcal{I}$ obtained from T' by replacing the leaf of b by an internal node with children a_i and a_j.

Proof: We will prove this by contradiction. Suppose $\hat{T}$ is optimal for $\mathcal{I}$. Then $\text{COST}(\hat{T}) < \text{COST}(T)$. In $\hat{T}$ we know a_i and a_j are siblings by Lemma 6.2.3. Now consider $\hat{T}'$ for instance $\mathcal{I}'$ where we merge a_i, a_j leaves and their parent into a leaf for symbol z.



Then

$$\text{COST}(\hat{T}') = \text{COST}(\hat{T}) - p(a_i) - p(a_j) < \text{COST}(T) - p(a_i) - p(a_j) = \text{COST}(T')$$

This contradicts the fact that T' is optimal binary encoding tree for $\mathcal{I}'$. Hence T is optimal. ■

5.2.2 Algorithm

Idea: We are going to build the tree up from the leaf level. We will take two characters x, y, and "merge" them into a single character, z, which then replaces x and y in the alphabet. The character z will have probability equal to the sum of x and y's probabilities. Then we continue recursively building the code on the new alphabet, which has one fewer character.

Since we always need the least frequent element and the second least frequent element we have to use the data structure called MIN-PRIORITY QUEUE. So the following algorithm uses a MIN-PRIORITY QUEUE Q keyed on the probabilities to identify the two least frequent objects.

Time Complexity: To create the priority queue it takes $O(n)$ time in line 4-5. Then for each iteration of the for loop in line 6 the EXTRACT-MIN operation takes $O(\log n)$ time and then to insert an element it also takes $O(\log n)$ time. Hence each iteration takes $O(\log n)$ time. Since the for loop has $n-1 = O(n)$ many iterations the running time for the algorithm is $O(n \log n)$.

Remark: We can reduce the running time to $O(n \log\log n)$ by replacing the binary min-heap with a van Emde Boas tree.

Algorithm 14: HUFFMAN-ENCODING(A, P)

Input: Set of n symbols $A = \{a_1, \ldots, a_n\}$ and their probabilities $P = \{p_1, \ldots, p_n\}$

Output: Optimal Binary Encoding $b : A \to \{0,1\}^*$ for A with minimum $\text{COST}(b) = \sum_{i=1}^{n} p(a_i) \cdot \text{LEN}(b(a_i))$.

```
1  begin
2      n ⟵ |A|
3      Q ⟵ MIN-PRIORITY QUEUE
4      for x ∈ A do
5          INSERT(Q, x)
6      for i = 1, ... n − 1 do
7          z ⟵ New internal tree node
8          x ⟵ EXTRACT-MIN(Q), y ⟵ EXTRACT-MIN(Q)
9          left[z] ⟵ x, right[z] ⟵ y
10         p(z) ⟵ p(x) + p(y)
11         INSERT(Q, z)
12     return Last element left in Q as root
```

Theorem 5.2.5 Correctness of Huffman's Algorithm

The above Huffman's algorithm produces an optimal prefix code tree

Proof: We will prove this by induction on n, the number of symbols. For base case $n = 1$. There is only one tree possible. For $n = k$ we know that by Lemma 6.2.3 and Lemma 6.2.1 that the two symbols x and y of lowest probabilities are siblings and they have the maximum height. Huffman's algorithm replaces these nodes by a character z whose probability is the sum of their probabilities. Now we have 1 less symbols. So by inductive hypothesis Huffman's algorithm computes the optimal binary encoding tree for the $k - 1$ symbols. Call it T_{n-1}. Then the algorithm replaces z with a parent node with children x and y which results in a tree T_n whose cost is higher by a fixed amount $p(z) = p(x) + p(y)$. Now since T_{n-1} is optimal by Theorem 6.2.4 we have T_n is also optimal. ■

5.3 Matroids