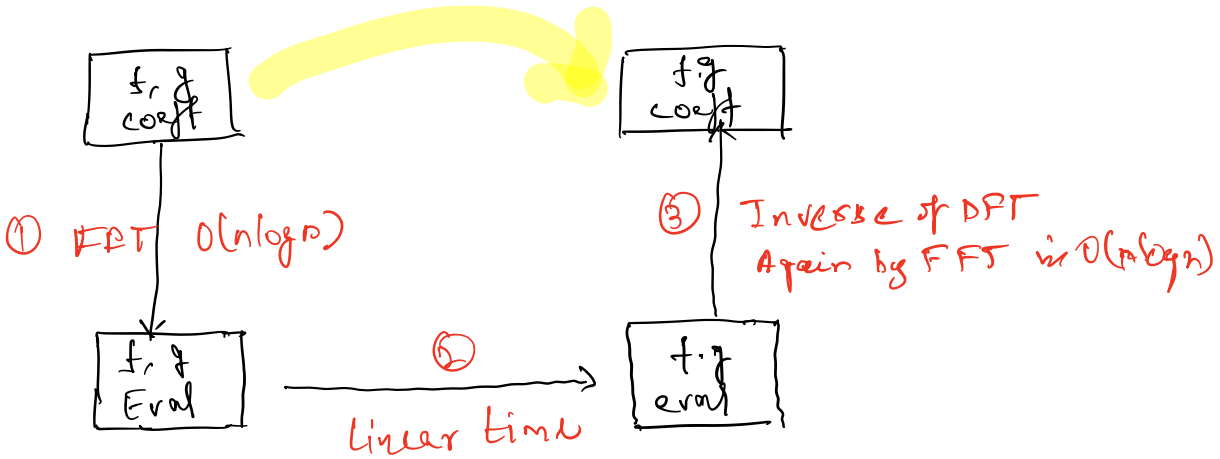
Plan

① Polynomial multiplication contd.

② start with Dynamic Programming.

## Poly Mult

Recall the overall structure




### Step 1

1o- Choose the points of evaluation carefully.

- pick $k \geq 2n$, powers of 2
- Evaluate at all $k^{th}$ roots of 1 in $\mathbb{C}$.

$$\{1, \omega_k, \omega_k^2, \cdots, \omega_k^{k-1}\} \qquad \omega_k = e^{i\frac{2\pi}{k}}$$

2o Evaluate recursively

- Define $f_{even}, f_{odd}$, deg $\leq \frac{n-1}{2}$

$$f_{even} = f_0 + f_2 x + f_4 x^2 + \cdots \checkmark$$
$$f_{odd} = f_1 + f_3 x + f_5 x^2 + \cdots \checkmark$$

- $f(x) = f_{even}(x^2) + x \cdot f_{odd}(x^2)$ — (1)
- Recursion.

Crucial prop: squares of $t^{th}$ roots of 1 give the set of all $t/2^{th}$ roots of 1

$$\{ \omega_t^{2i} : i \in \{0,1,\ldots,t-1\} \}$$

$$= \{ \omega_{t/2}^{j} : j \in \{0,1,\ldots,\tfrac{t}{2}-1\} \}$$

So eqn (1) reduces the prob of eval. a deg $n-1$ poly on all $t^{th}$ roots of 1 to 2 instances of eval. deg $\frac{n-1}{2}$ polys on $t/2^{th}$ roots of 1.

$+ O(n)$ time post processing

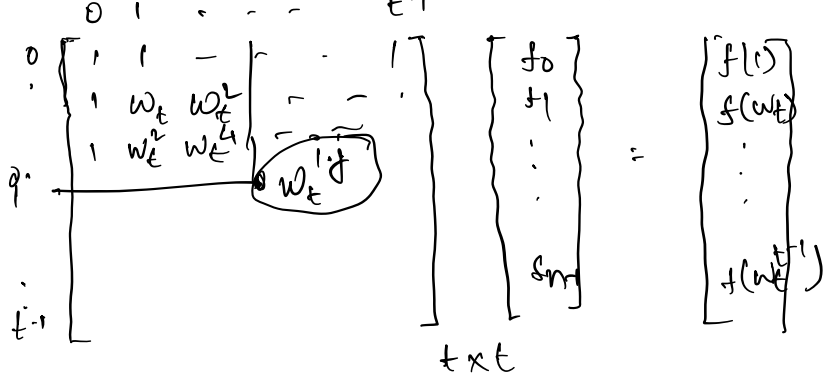
$$T(n) \le 2T(n/2) + O(n).$$

$$\Rightarrow T(n) \le O(n \log n).$$

---

Sum:

$O(n \log n)$ algo to evaluate $f$ on $\{1, \omega_t, \omega_t^2, \ldots, \omega_t^{t-1}\}$.



Step 3: Evaluation representation to coefficients.

Given: (Eval of f.g on $t^{th}$ roots of unity)

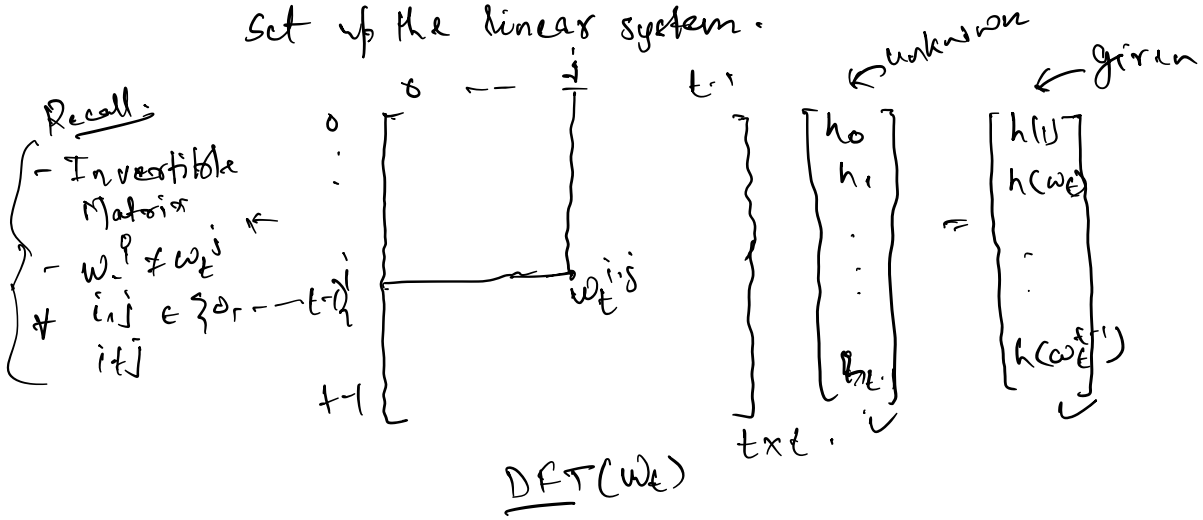
Want: coeff representation of f.g.

Assume: $h(x)$ of deg $\leq t-1$ and we have evaluations of $h$ on $t^{th}$ roots of unity.

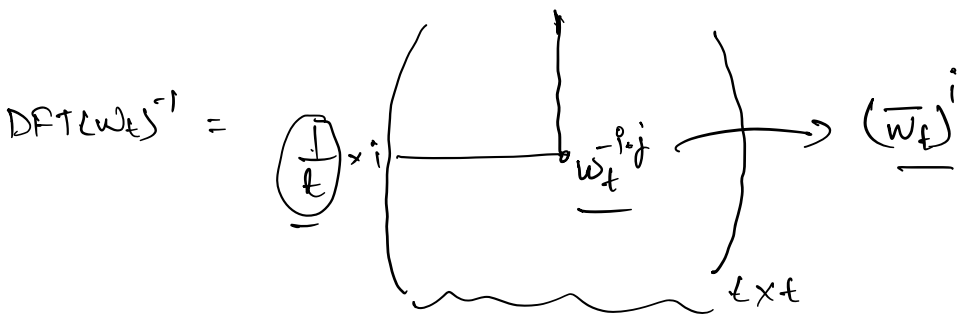
Want: coefficients of $h$.

Let $h(x) := h_0 + h_1 x + h_2 x^2 + \cdots + h_{t-1} \cdot x^{t-1}$.

Set up the linear system.




$\Rightarrow$ coeff vector of $h$ = $DFT^{-1}(w_t)$ · Evaluation vector.



Rephrasing this problem as an instance of step (1).

Step 1 – Given the coeff vector of a poly, evaluate it at all $t^{th}$ roots of 1.

– Want it fast.

Here:

$$DFT(\omega_t^{-1}) = \frac{1}{t} \cdot M$$

$$M_{i,j} = \omega_t^{-i \cdot j}$$

Want: $M \cdot \begin{pmatrix} h(1) \\ h(\omega_t) \\ \vdots \\ h(\omega_t^{t-1}) \end{pmatrix}$ Fast

$\omega_t^{-1} = \bar{\omega}_t$

Let $\tilde{H}(x) = h(1) + x \cdot h(\omega_t) + x^2 \cdot h(\omega_t^2) + \cdots + x^{t-1} \cdot h(\omega_t^{t-1})$

$M \cdot coeff(H) =$ Evaluations of $\tilde{H}$ on $\{1, \omega_t^{-1}, \omega_t^{-2}, \ldots, \omega_t^{-(t-1)}\}$

– If we set $= \{1, \omega_t, \omega_t^2, \ldots, \omega_t^{t-1}\}$

then FFT

Fact: $\{1, \omega_t^{-1}, \omega_t^{-2}, \ldots, \omega_t^{-(t-1)}\} = \{1, \omega_t, \omega_t^2, \ldots, \omega_t^{(t-1)}\}$

So, done via FFT.

Proof: homework.

---

## Dynamic Programming

- Clever Recursion
- Recursion with memory

Ex1: computing Fibonacci numbers

$F_1 = 1$

$F_2 = 1$

$\forall n > 2,\ F_n = F_{n-1} + F_{n-2}$

Q: An algorithm for computing $F_n$.

Algo1 - Recursion

1) if $n \in \{1, 2\}$ return 1
2) else Return $F_{n-1} + F_{n-2}$.

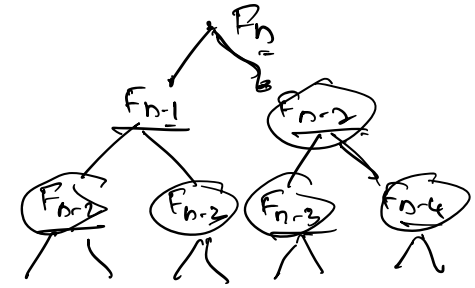
Running time: $T(n) \leq T(n-1) + T(n-2) + O(1)$

Recursion tree:

Binary tree of depth $\geq n/2$ ✓

$\Rightarrow 2^{n/2}$ vertices in the tree.

$\Omega(n)$ depth




Observation:

Every recursive call is for $\{F_1, F_2, \ldots, F_{n-1}\}$

Each $F_i$ is called multiple times.

- Stop the repetitive computation.
- Remember what we compute.

Algo 2

$F[i] = F_i$

1. $F[1:n]$ — global array
2. $F[1] = 1$, $F[2] = 1$
3. for $i = 3$ to $n$

   set $F[i] := F[i-1] + F[i-2]$.

Running time = $O(n)$.

Ex 2:

Computing binomial coefficients.

$$\binom{n}{i} = \frac{n!}{i!\,(n-i)!} \equiv$$ # of ways of choosing k-distinct of objects from n distinct objects.

Recursion:

Pascal's identity

$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$$

HW: prove this identity.

$$\binom{m}{0} = 1 \quad \forall m \geq 0$$

$$\binom{m}{j} = 0 \quad \text{if } j > m$$

Input: $n, k \in \mathbb{N}$

Output: $\binom{n}{k}$.

- Obvious recursive algo. ——
- Again an exponential size recursion tree.
- But all recursive calls are to compute

$$\left\{ \binom{m}{j} : 0 \leq m \leq n,\ 0 \leq j \leq k \right\}$$ — size $O(nk)$

- Can again do this more efficiently by remembering the things already computed.
  - Set up a $n \times k$ matrix $M$ $\quad \left[ M(i,j) = \binom{i}{j} \right]$
  - Base cases. —
  - For $i = 0$ to $n$
    For $j = 0$ to $k$

$M(i,j) = M(i,j-1) + M(i-1,j-1)$.

Running time: $O(nk)$.

Divide and Conquer | Dynamic Prog

Both rely on recursive structure

- smaller instances are gen. straightforward
  - Divide an array in middle
  - split a polynomials
- typically disjoint subproblems
- Analysis relies on some recurrence, Master thm.

- smaller instances might need more care
- non-disjoint smaller instances
- smaller instances come from a small set.

DP: Recursive Substructure + Large overlap / Not too many subproblems.

# Subset Sum

Input: positive integers $a_1, \ldots, a_n, B$

Output: Decide if there is a subset $S \subseteq \{1, 2, \ldots, n\}$ s.t. $\sum_{i \in S} a_i = B$.

(Return such a subset $S$ if it exists)

Algo I:

- Iterate over all subsets of $\{1, 2, \ldots, n\}$
- check whether sum of numbers in that set equals $B$.

Running time: $\text{poly}(n) \cdot 2^n$. $\quad O(n^2 \cdot 2^n)$

- can we do this faster?

High level intuition: suppose there is a 'right' subset $S$:

Notation:
SubSum$(a_1, \ldots, a_n; B)$ = yes/true iff $\exists\, S \subseteq \{1, 2, \ldots, n\}$ s.t. $\sum_{i \in S} a_i = B$

Consider $a_n$.

Case 1: $a_n \in S$ ⟹ I: Subset Sum $(a_1, \ldots, a_{n-1}; B - a_n)$ is True.

Case 2: $a_n \notin S$

$I_2$: subsetSum($a_1, \ldots, a_{n-1}$; B) } is True.

Claim: The original instance is True iff at least one of the $I_1, I_2$ is True. } HW.