

Divide and conquer 3

Linear time sorting

CS240

Spring 2024

Rui Fan

Polynomial multiplication

- Let $A(x) = \sum_{j=0}^{n-1} a_j x^j$ and $B(x) = \sum_{j=0}^{n-1} b_j x^j$ be two polynomials.
- Compute the product $C(x) = A(x)B(x)$.

$$\begin{array}{r}
 6x^3 + 7x^2 - 10x + 9 \quad A(x) \\
 - 2x^3 \quad B(x) \\
 \hline
 - 30x^3 - 35x^2 + 50x - 45 \\
 24x^4 + 28x^3 - 40x^2 + 36x \\
 - 12x^6 - 14x^5 + 20x^4 - 18x^3 \\
 \hline
 - 12x^6 - 14x^5 + 44x^4 - 20x^3 - 75x^2 + 86x - 45
 \end{array}$$

- $C(x)$'s degree is at most $2n-2$.
- $C(x) = \sum_{j=0}^{2n-2} c_j x^j$, where $c_j = \sum_{k=0}^j a_k b_{j-k}$.
- The naive method takes $O(n^2)$ time.
- Using **FFT** and divide & conquer, we'll do it in $O(n \log n)$ time.

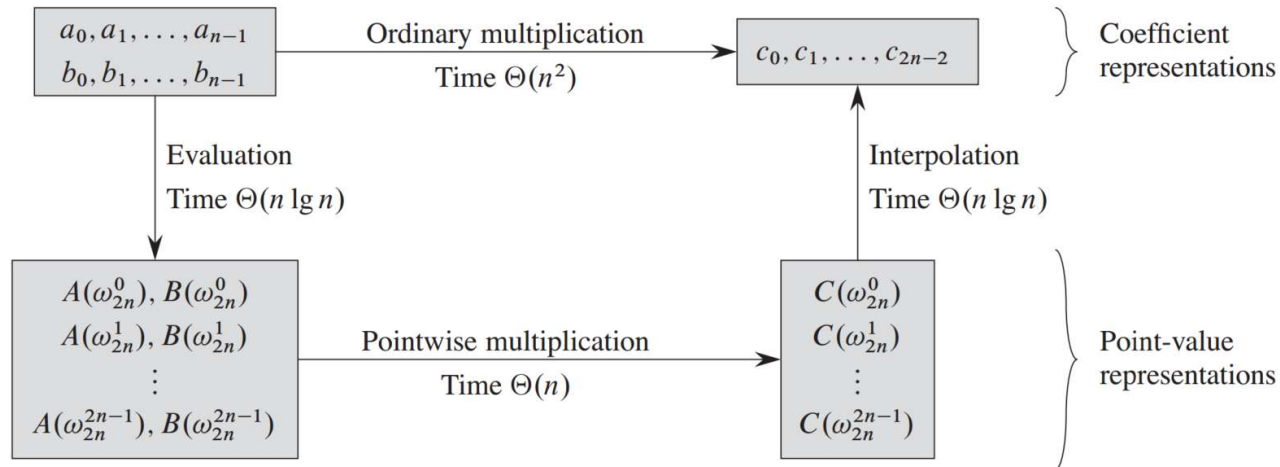
Polynomial representations

- The **coefficient representation** of a polynomial is $A(x) = \sum_{j=0}^{n-1} a_j x^j$.
- The **point-value representation** is $\{(x_0, y_0), (x_1, y_1), \dots, (x_{n-1}, y_{n-1})\}$, where $y_k = A(x_k)$.
- If we have $A(x), B(x)$ in point-value form, computing $C(x) = A(x)B(x)$ can be done in **$O(n)$ time**.
 - Pick $2n$ points x_0, \dots, x_{2n-1} . Then $C(x_k) = A(x_k)B(x_k)$ for $k = 0, \dots, 2n - 1$.
 - In coefficient form, naive multiplication takes $O(n^2)$ time.
- Given $A(x)$ represented in point-value form, we can **reconstruct** the coefficient representation.

$$\begin{pmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \cdots & x_1^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n-1} & x_{n-1}^2 & \cdots & x_{n-1}^{n-1} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_{n-1} \end{pmatrix}$$

- Matrix V is the Vandermonde matrix. It's nonsingular and **invertible**.
 - So $a = V^{-1}y$.
 - Can compute a more efficiently than general matrix inversion and multiplication.

Fast polynomial multiplication

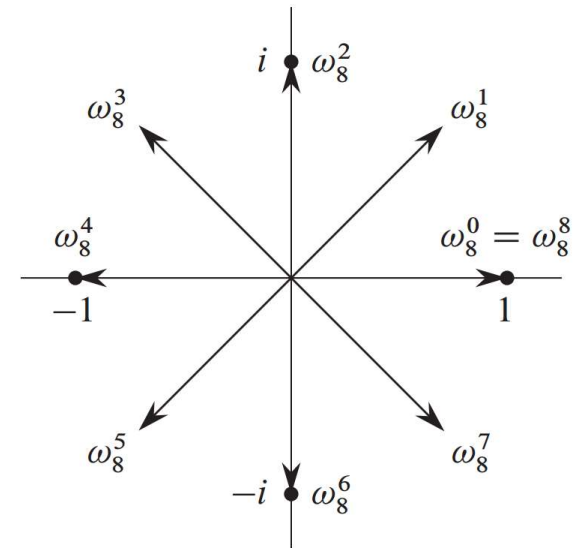


Source: Introduction to Algorithms, Cormen et al

- We want to multiply polynomials A and B quickly.
- **Evaluate** A and B at $2n$ points to get point-value representations.
 - Evaluation can be at any points. But we'll do it at the **$2n$ 'th roots of unity**.
 - Evaluating A and B at $2n$ 'th roots of unity by **FFT** takes $O(n \log n)$ time.
- **Pointwise multiply** A and B to get point-value representation for C .
 - This takes $O(n)$ time.
- **Transform** point-value representation of C to coefficient form.
 - This is done using **inverse FFT** in $O(n \log n)$ time.

Roots of unity

- An **n 'th root of unity** is a complex number ω s.t. $\omega^n = 1$.
- There are n n 'th roots of unity, and they have the form $e^{\frac{2\pi i k}{n}}$, for $0 \leq k \leq n - 1$.
 - Write $\omega_n = e^{\frac{2\pi i}{n}}$, so that the n 'th roots of unity are $\omega_n^0, \omega_n^1, \dots, \omega_n^{n-1}$.
- Below, assume n is a power of 2.
- **Fact 1** $(\omega_n^k)^n = 1$ for any $0 \leq k \leq n - 1$.
- **Fact 2** $\omega_n^{k+\frac{n}{2}} = -\omega_n^k$.
- **Fact 3** $(\omega_n^k)^2 = \omega_n^{2k} = \omega_{n/2}^k$.
- **Fact 4** For any $0 \leq k \leq n - 1$, $\sum_{j=0}^{n-1} (\omega_n^k)^j = 0$.

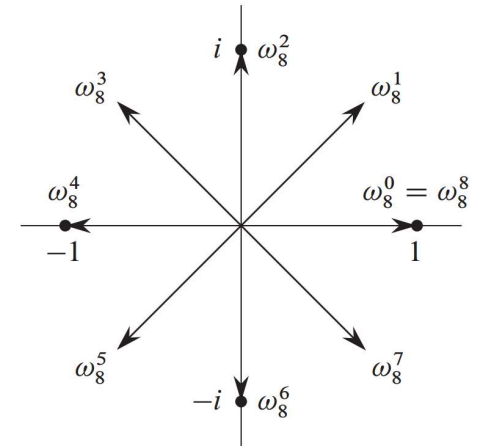




Discrete Fourier Transform

- Given a degree $n - 1$ polynomial $A(x) = a_0 + a_1x + a_2x^2 + \dots + a_{n-1}x^{n-1}$, DFT computes $A(\omega_n^0), A(\omega_n^1), \dots, A(\omega_n^{n-1})$.
- FFT is a fast algorithm for DFT that runs in $O(n \log n)$ time using divide and conquer.
 - Assume n is a power of 2.
- FFT has a vast number of applications in CS and EE.
- One of IEEE's top 10 most important algorithms of the 20th century.
 - Popularized by Cooley and Tukey in 1965.
 - But variants known to Gauss in 1805!

Fast Fourier Transform



- Let $A^{[0]} = a_0 + a_2x^1 + a_4x^2 + \dots + a_{n-2}x^{\frac{n}{2}-1}$
 $A^{[1]} = a_1 + a_3x^1 + a_5x^2 + \dots + a_{n-1}x^{\frac{n}{2}-1}$.
- Then $A(x) = A^{[0]}(x^2) + xA^{[1]}(x^2)$.
- So can compute $A(\omega_n^0), A(\omega_n^1), \dots, A(\omega_n^{n-1})$ by computing $A^{[0]}$ and $A^{[1]}$ at $(\omega_n^0)^2, (\omega_n^1)^2, \dots, (\omega_n^{n-1})^2$, and multiplying some terms by $\omega_n^0, \omega_n^1, \dots, \omega_n^{n-1}$.
- But $(\omega_n^{k+n/2})^2 = \omega_n^{2k+n} = \omega_n^{2k} = (\omega_n^k)^2$ by Fact 1.
 - So $\{(\omega_n^0)^2, (\omega_n^1)^2, \dots, (\omega_n^{n-1})^2\} = \{(\omega_n^0)^2, (\omega_n^1)^2, \dots, (\omega_n^{n/2-1})^2\}$, i.e. only need to evaluate $A^{[0]}$ and $A^{[1]}$ at $n/2$ points instead of n !
- Also, $(\omega_n^k)^2 = \omega_n^{2k} = \omega_{n/2}^k$.
 - So $\{(\omega_n^0)^2, (\omega_n^1)^2, \dots, (\omega_n^{n/2-1})^2\} = \{\omega_{n/2}^0, \omega_{n/2}^1, \dots, \omega_{n/2}^{n/2-1}\}$, i.e. only need to evaluate $A^{[0]}$ and $A^{[1]}$ at the $(n/2)$ 'th roots of unity.

Fast Fourier Transform

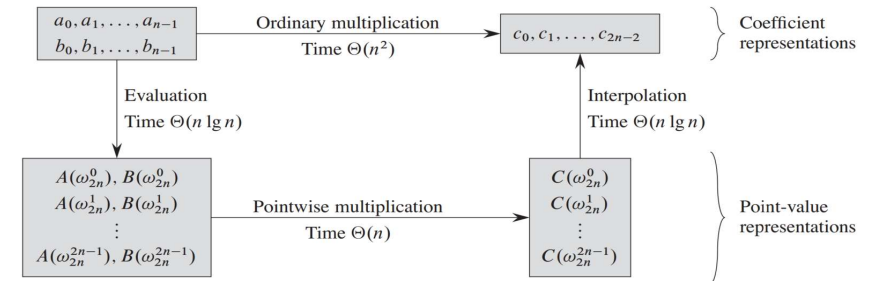
- Thus, computing $A(x) = A^{[0]}(x^2) + xA^{[1]}(x^2)$ for $x \in \{\omega_n^0, \omega_n^1, \dots, \omega_n^{n-1}\}$ requires
 - Computing for $A^{[0]}(x)$ and $A^{[1]}(x)$ for $x \in \{\omega_{n/2}^0, \omega_{n/2}^1, \dots, \omega_{n/2}^{n/2-1}\}$.
 - These are **also DFT's**, so can be done recursively using two $n/2$ -point FFT's.
- For $0 \leq k \leq \frac{n}{2} - 1$
 - $A(\omega_n^k) = A^{[0]}(\omega_{n/2}^k) + \omega_n^k A^{[1]}(\omega_{n/2}^k)$
 - $A(\omega_n^{k+n/2}) = A^{[0]}(\omega_{n/2}^{k+n/2}) + \omega_n^{k+n/2} A^{[1]}(\omega_{n/2}^{k+n/2})$
 $= A^{[0]}(\omega_{n/2}^k) - \omega_n^k A^{[1]}(\omega_{n/2}^k)$

FFT algorithm

```
FFT(n, a0, a1, ..., an-1) {  
    if (n == 1) return a0  
  
    (e0, e1, ..., en/2-1) ← FFT(n/2, a0, a2, a4, ..., an-2)  
    (d0, d1, ..., dn/2-1) ← FFT(n/2, a1, a3, a5, ..., an-1)  
  
    for k = 0 to n/2 - 1 {  
        ωk ← e2πik/n  
        yk ← ek + ωk dk  
        yk+n/2 ← ek - ωk dk  
    }  
    return (y0, y1, ..., yn-1)  
}
```

- Let $T(n)$ be the time to compute a size n FFT.
Then $T(n) = 2T\left(\frac{n}{2}\right) + O(n)$, so $T(n) = \Theta(n \log n)$.

Inverse FFT



- So far we have
 - Computed A, B at the $2n$ 'th roots of unity.
 - Pointwise multiplied A, B to get point-value representation of C.
- To multiply A and B, the last step is to convert C back to coefficient representation using **inverse FFT**.
- Inverse FFT takes a polynomial represented in point-value form and computes the polynomial's coefficient form.
 - I.e. given $[y_0 \ y_1 \ \dots \ y_{n-1}]^T$ below, it computes $[a_0 \ a_1 \ \dots \ a_{n-1}]^T$.

$$\begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 1 & \dots & 1 \\ 1 & \omega_n & \omega_n^2 & \omega_n^3 & \dots & \omega_n^{n-1} \\ 1 & \omega_n^2 & \omega_n^4 & \omega_n^6 & \dots & \omega_n^{2(n-1)} \\ 1 & \omega_n^3 & \omega_n^6 & \omega_n^9 & \dots & \omega_n^{3(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_n^{n-1} & \omega_n^{2(n-1)} & \omega_n^{3(n-1)} & \dots & \omega_n^{(n-1)(n-1)} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_{n-1} \end{pmatrix}$$

- For the polynomial multiplication problem, the a vector represents C .
 - $a = V_n^{-1}y$.
- Need to know the inverse of Vandermonde matrix V_n .

Inverse FFT

- **Thm** For $j, k = 0, \dots, n-1$, the (j, k) entry of V_n^{-1} is ω_n^{-kj}/n .

$$V_n^{-1} = \frac{1}{n} \begin{bmatrix} 1 & 1 & 1 & 1 & \dots & 1 \\ 1 & \omega^{-1} & \omega^{-2} & \omega^{-3} & \dots & \omega^{-(n-1)} \\ 1 & \omega^{-2} & \omega^{-4} & \omega^{-6} & \dots & \omega^{-2(n-1)} \\ 1 & \omega^{-3} & \omega^{-6} & \omega^{-9} & \dots & \omega^{-3(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{-(n-1)} & \omega^{-2(n-1)} & \omega^{-3(n-1)} & \dots & \omega^{-(n-1)(n-1)} \end{bmatrix}$$

- **Proof**

$$[V_n^{-1}V_n]_{jj'} = \sum_{k=0}^{n-1} \left(\frac{\omega_n^{-kj}}{n} \right) \left(\omega_n^{kj'} \right) = \frac{1}{n} \sum_{k=0}^{n-1} \omega_n^{k(j'-j)}$$

- By Fact 4, (j, j') entry is 1 exactly when $j = j'$, and 0 otherwise. So $V_n^{-1}V_n$ is the identity matrix.

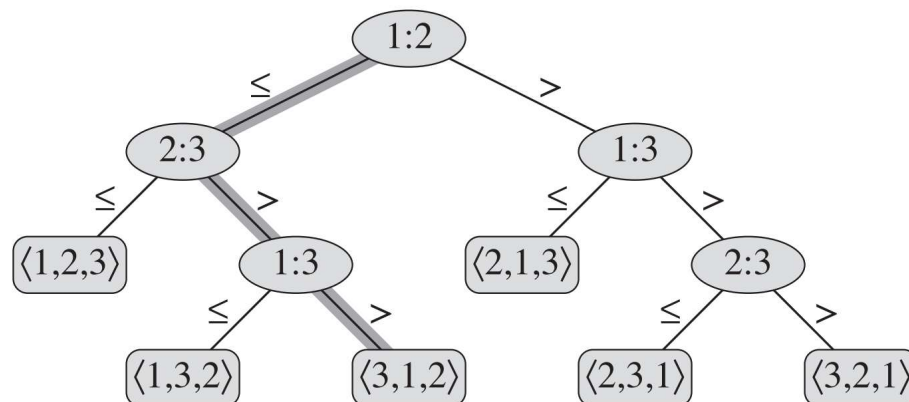
Inverse FFT

$$V_n^{-1} = \frac{1}{n} \begin{bmatrix} 1 & 1 & 1 & 1 & \dots & 1 \\ 1 & \omega^{-1} & \omega^{-2} & \omega^{-3} & \dots & \omega^{-(n-1)} \\ 1 & \omega^{-2} & \omega^{-4} & \omega^{-6} & \dots & \omega^{-2(n-1)} \\ 1 & \omega^{-3} & \omega^{-6} & \omega^{-9} & \dots & \omega^{-3(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{-(n-1)} & \omega^{-2(n-1)} & \omega^{-3(n-1)} & \dots & \omega^{-(n-1)(n-1)} \end{bmatrix}$$

- Since $[a_0 \ a_1 \ \dots \ a_{n-1}]^T = V_n^{-1} [y_0 \ y_1 \ \dots \ y_{n-1}]$, we have $a_j = \frac{1}{n} \sum_{k=0}^{n-1} y_k \omega_n^{-kj}$.
- Consider the degree $n - 1$ polynomial $Y(x) = \frac{1}{n} (y_0 + y_1 x + y_2 x^2 + \dots + y_{n-1} x^{n-1})$.
 - We want to compute Y at $1, \omega_n^{-1}, \omega_n^{-2}, \dots, \omega_n^{-n}$.
- This is just another DFT, which we can do in $O(n \log n)$ time.
- Thus, we can multiply two degree $n - 1$, or more generally, do $n - 1$ point convolution in $O(n \log n)$ time.

Comparison sorts

- In insertion sort, mergesort and Quicksort, the output order only depended on **comparisons** between the input values.
 - **Ex** In insertion sort, we compare a value with other values to determine its sorted position.
 - These algorithms are comparison sorts.
 - They all take $\Omega(n \log n)$ time to sort n inputs.
- There is a general **$\Omega(n \log n)$ lower bound** on the time complexity of **any comparison sort** algorithm.
 - Any algorithm in which the output is only determined by comparisons between input values takes at least $\Omega(n \log n)$ steps.





Beyond comparison sorts

- To sort faster than $\Omega(n \log n)$ time, we need to use **other operations** besides comparison.
- An algorithm can sort in $O(n)$ linear time by reading the value of inputs, and using these for array indexing, comparing digits, etc.
- $O(n)$ time is asymptotically the best possible, since we need to read all n inputs.
- We'll look at counting and radix sort.



Counting sort

- Counting sort assumes all input values are **integers** in the **range 0 to k**, for some k.
- The algorithm runs in $\Theta(n)$ time when **k is small**.
 - As k gets larger, the algorithm becomes increasingly inefficient.
 - Counting sort is used as a stand-alone algorithm, and also as a subroutine in other algorithms, e.g. radix sort.
 - Radix sort ensures k is small, so counting sort is fast.

Counting sort

- Since we assume all inputs are in $[0, k]$, we use a size k array C to store **how many inputs** have each value.
 - $C[i] = c$ if there are c inputs with value i .
- Iterate through input array A .
 - For input value $A[i]$, increment $C[A[i]]$ to record an additional occurrence of value $A[i]$.
- Once we know number of occurrences of each value, we know the sorted output.
 - **Ex** If $C = [2, 1, 3, 0, 0, 1]$, then output is 0,0,1,2,2,2,5.
- Value i occurs $C[i]$ times.
 - It appears after values $0, 1, \dots, i - 1$.
 - There are $\sum_{j=0}^{i-1} C[j]$ values $0, 1, \dots, i - 1$.
 - So first occurrence of i is at index $\sum_{j=0}^{i-1} C[j] + 1$, and last occurrence is at $\sum_{j=0}^i C[j]$.
- After computing count array C , compute **prefix sum** C' of C .
 - $C'[i] = \sum_{j=0}^i C[j]$. Also, set $C'[-1] = 0$.
 - Value i occurs in indices $[C'[i - 1] + 1, C'[i]]$ of output.

Example

	1	2	3	4	5	6	7	8
A	2	5	3	0	2	3	0	3
	0	1	2	3	4	5		
C	2	0	2	3	0	1		

(a)

	0	1	2	3	4	5
C	2	2	4	7	7	8

(b)

	1	2	3	4	5	6	7	8
B							3	
	0	1	2	3	4	5		
C	2	2	4	6	7	8		

(c)

	1	2	3	4	5	6	7	8
B		0					3	
	0	1	2	3	4	5		
C	1	2	4	6	7	8		

(d)

	1	2	3	4	5	6	7	8
B		0				3	3	
	0	1	2	3	4	5		
C	1	2	4	5	7	8		

(e)

	1	2	3	4	5	6	7	8
B	0	0	2	2	3	3	3	5

(f)

Source: Introduction to Algorithms, Cormen et al.

- In (a), C contains number of occurrences of each input value in A .
- In (b), compute the prefix sum of C .
- In (c)-(f), iterate through A in reverse order. 为了stable
 - Put value $A[i]$ in position $C[A[i]]$ of output, then decrement $C[A[i]]$ (because we output one more copy of $A[i]$).

Pseudocode and complexity

COUNTING-SORT(A, B, k)

```
1  let  $C[0..k]$  be a new array
2  for  $i = 0$  to  $k$ 
3       $C[i] = 0$ 
4  for  $j = 1$  to  $A.length$ 
5       $C[A[j]] = C[A[j]] + 1$ 
6  //  $C[i]$  now contains the number
7  for  $i = 1$  to  $k$ 
8       $C[i] = C[i] + C[i - 1]$ 
9  //  $C[i]$  now contains the number
10 for  $j = A.length$  downto 1
11      $B[C[A[j]]] = A[j]$ 
12      $C[A[j]] = C[A[j]] - 1$ 
```

- First for loop resets all counts.
 - $O(k)$ time.
- Second for loop counts occurrences of each value.
 - $O(n)$ time.
- Third loop computes prefix sum.
 - $O(k)$ time.
- Last for loop uses prefix sum to scatter inputs to output positions.
 - $O(n)$ time.
- Overall complexity $\Theta(n + k)$.
- If $k = O(n)$, then complexity is $\Theta(n)$.



Stability

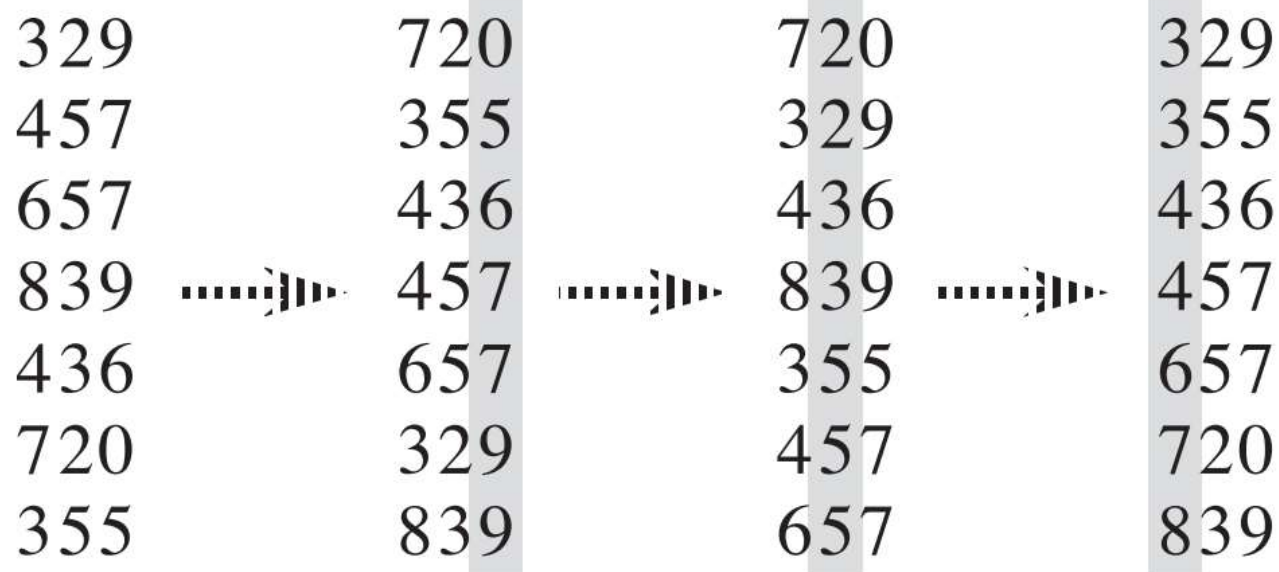
- A useful property of counting sort is that it's **stable**.
 - If two inputs are equal, then their order in the output is the **same** as their order in the input.
 - This is why we iterated through A in **reverse order** when producing B.
- **Ex** If input is $4, 1_A, 5, 2_A, 1_B, 2_B, 1_C$, then output is $1_A, 1_B, 1_C, 2_A, 2_B, 4, 5$.
- Stability is necessary when counting sort is used as a subroutine in other sorts, e.g. radix sort.



Radix sort

- Sort **digit by digit**, from least to most significant digit.
- Take list of input values, sort them on the singles digit.
- Take the new list, sort values on the tens digit.
- Take the new list, sort values on hundreds digit.
Etc.
- The sorting algorithm for each digit must be **stable**.
- We will use counting sort.
 - It's stable.
 - Since we sort a digit at a time, the values being sorted are between 0 and 9.
 - Sorting n inputs takes $O(n)$ time.

Example



Notice due to stability, after sorting by the 10's digit, 436 and 839 (for example) keep the same order they had after sorting by the 1's digit.



Correctness

- **Lemma 1** Let x and y be two inputs to radix sort.

- Let k be the most significant digit on which they differ.
- Suppose k 'th digit of x is less than k 'th digit of y .

After sorting the k 'th digit (in nondecreasing order), x will come before y in the **remainder** of the execution.

- **Proof** x comes before y right after sorting the k 'th digit.

- x and y are equal on all higher digits.
- So when sorting on higher digits, x and y are always tied.
- Since the sort is stable, x stays before y from the k 'th sort onwards.



Complexity

- **Lemma 2** Suppose we sort n d -digit numbers, where each digit is between 0 to $k - 1$. Then radix sort takes $O(d(n + k))$ time.
- **Proof** Since each digit is between 0 and $k - 1$, then counting sort takes $O(n + k)$ time per digit. So the total time is $O(d(n + k))$.
- **Ex** Sorting n d -digit binary numbers takes $O(dn)$ time.



Complexity

- **Lemma 3** Given n b -bit numbers and $r \leq b$.
Radix sort takes $O\left(\frac{b}{r}(n + 2^r)\right)$ time.
- **Proof** Break the b bits into blocks of r digits, having values between 0 and $2^r - 1$.
 - **Ex** For $b = 6, r = 2$, break the value 100111 into blocks 10, 01 and 11.
 - There are $d = \lceil b/r \rceil$ such blocks.
 - We can think of each b -bit number as a d digit number, where each digit has value between 0 and $2^r - 1$.
 - The lemma follows from Lemma 2.



Complexity

- **Lemma 4** Setting $r = \min(\lfloor \log n \rfloor, b)$ minimizes the running time $\Theta\left(\frac{b}{r}(n + 2^r)\right)$.
- **Proof** If $b < \lfloor \log n \rfloor$, then for any $r \leq b$, we have $n + 2^r = \Theta(n)$. So we set $r = b$ to minimize b/r .
 - If $b \geq \lfloor \log n \rfloor$, setting $r = \lfloor \log n \rfloor$ makes running time $\Theta\left(\frac{bn}{\log n}\right)$.
 - We show $r > \lfloor \log n \rfloor$ and $r < \lfloor \log n \rfloor$ both result in slower running times.
 - If $r > \lfloor \log n \rfloor$, then $2^r > n$, and 2^r in numerator increases faster than r in denominator, so running time increases.
 - If $r < \lfloor \log n \rfloor$, then the b/r term increases, but the $n + 2^r$ remains $\Theta(n)$.
- In other words, radix sort is efficient when there are many short numbers, but not when there are a few long numbers.